NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

DESIGN RECOVERY AND IMPLEMENTATION OF THE AYK-14 VHSIC PROCESSOR MODULE ADAPTER WITH FIELD PROGRAMMABLE GATE ARRAY TECHNOLOGY

by

Bryan J. Fetter

December 2002

Thesis Advisor: Russell W. Duren
Second Reader: Hersch Loomis

Approved for public release; distribution is unlimited.

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			*Form Approved OMB No. 0704-0188*
1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** December 2002	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE: Design Recoverey and Implementation of the AYK-14 VHSIC Processor Module Adapter with Field Programmable Gate Array Technology			**5. FUNDING NUMBERS**
6. AUTHOR(S) Fetter, Bryan James			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT **Approved for public release; distribution is unlimited.**			**12b. DISTRIBUTION CODE**

13. ABSTRACT *(maximum 200 words)*

The rapid pace of change in the electronics industry and the significant reduction in military budgets over the past decade has forced many military aircraft to extend their service lifetimes. This has led to aircraft with outdated avionics systems being required to accomplish new and more complex missions. This thesis examines the process of reengineering an outdated avionics system to economically upgrade its capabilities through the FPGA implementation of a binary compatible replacement. The system targeted is the AN/AYK-14(V) Navy Standard Airborne Computer, specifically the XN-8 chassis used as the mission computer onboard the F/A-18 C/D aircraft. This thesis is also intended to provide a resource document on the AYK-14 for a study being conducted by the Naval Air Systems Command (NAVAIR) Advanced Weapons Laboratory (AWL). The design of the Input / Output module of the VHSIC Processor Module was recovered through research of documentation and hardware testing. The recovered design was modeled using VHDL, synthesized and implemented using computer-aided (CAD) design tools. This thesis shows that replacement of legacy systems through use of FPGA technology is a viable option, however, expansion of the design is recommended to create a truly binary compatible replacement.

14. SUBJECT TERMS Obsolescence, Legacy, FPGA, VHDL,VHSIC, Xilinx, SDRAM, AYK-14, Mil-Std-1553, State Machine, AVNET, Bus Controller, Data Bus, Software Interrupts, Reengineering, Design Recovery			**15. NUMBER OF PAGES** 218
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

DESIGN RECOVERY AND IMPLEMENTATION OF THE AYK-14 VHSIC PROCESSOR MODULE ADAPTER WITH FIELD PROGRAMMABLE GATE ARRAY TECHNOLOGY

Bryan J. Fetter
Lieutenant, United States Navy
B.S. Aerospace Engineering, United States Naval Academy, 1993

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN AERONAUTICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
DECEMBER 2002

Author: Bryan J. Fetter

Approved by: Russell W. Duren
Thesis Advisor

Hersch Loomis
Second Reader

Max Platzer
Chairman, Department of Aeronautics and Astronautics

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

The rapid pace of change in the electronics industry and the significant reduction in military budgets over the past decade has forced many military aircraft to extend their service lifetimes. This has led to aircraft with outdated avionics systems being required to accomplish new and more complex missions. This thesis examines the process of reengineering an outdated avionics system to economically upgrade its capabilities through the FPGA implementation of a binary compatible replacement. The system targeted is the AN/AYK-14(V) Navy Standard Airborne Computer, specifically the XN-8 chassis used as the mission computer onboard the F/A-18 C/D aircraft. This thesis is also intended to provide a resource document on the AYK-14 for a study being conducted by the Naval Air Systems Command (NAVAIR) Advanced Weapons Laboratory (AWL). The design of the Input / Output module of the VHSIC Processor Module was recovered through research of documentation and hardware testing. The recovered design was modeled using VHDL, synthesized and implemented using computer-aided design (CAD) tools. This thesis shows that replacement of legacy systems through use of FPGA technology is a viable option, however, expansion of the design is recommended to create a truly binary compatible replacement.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would like to thank Professor Russ Duren for providing me with the opportunity, means, and guidance to complete this thesis. His instruction and mentoring has extended well beyond this thesis and I am truly grateful for his friendship and support. I would also like to thank Professor Hersch Loomis for his instruction during the design process and for his support. I would like to also thank Mr. Rex Coombs, PMA-209, for his time and exceptional level of support. The use of his lab, the supply of numerous documents, and the loan of an AYK-14 and CCU were essential factors in completing this thesis.

I am also extremely grateful to the U.S. Naval Test Pilot School for providing me with the opportunity and means to complete this thesis. I would specifically like to thank CDR Rich Brasel for allowing the completion of my thesis to be my primary duty. I would like to thank CDR Paul Sohl for his guidance and support during this difficult career transition. In addition, I would like to thank CDR C.J. Junge for his friendship, support, and inspiration.

My extreme thanks also goes to CDR Mike ‘Croc’ Croskrey for being my teammate on the AYK-14 recovery. I am indebted to you for your support on this thesis and on my transition from the Navy. It is a privilege to have been your classmate and your friend.

And finally my thanks goes to my wife and best friend, Michelle. I can never thank you enough for your unwavering support. I could never have completed this thesis without you by my side. To reach the stars, you must stand on the shoulders of giants. Thank you for being my giant!

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. THE LEGACY AVIONICS ISSUE

The 1990's was a decade that ushered in many dramatic changes in the world. These changes had a profound effect on the U.S. government and the armed forces in particular. The two events that had the greatest effect on the military were the fall of communism and the technological revolution in the electronics industry.

The end of the Cold War left the military without a formidable adversary. This, in turn, led to budgetary changes that affected all branches of the military. More specifically, the funding for the acquisition of new military aircraft was greatly reduced. This occurred in parallel with a similar reduction in the budgets for modernization of existing, or 'legacy'[1], aircraft. In order to deal with the shrinking budget, the operational lifetimes of many of these legacy aircraft were extended beyond their original service lifetimes. This has led to the average age of a U.S. Military aircraft being 20 years and continuing to increase.[Ref. 2:p. 1]

This increase in average age has reduced the effectiveness and readiness of the armed forces as a whole. According to the 'Committee on Aging Avionics in Military Aircraft', the U.S. Air Force reported a 10 percent decrease in readiness during the 1990's. The committee attributed this decline to the increasing age of the aircraft, "particularly the aging avionics systems on which they depend." [Ref. 2:p. 1] The shrinking budgets for upgrades to these avionics mean that the decline in readiness will most likely continue unless lower cost solutions can be found.

The technological revolution that has occurred during the 1980's and 1990's has brought with it great advances in electronics and computing. However, the economic impetus behind these advances has increasingly come from the commercial sector. As Reference 2 states "whereas the military once provided a large and profitable market for the electronics industry, the military electronics market today constitutes less than 1 percent of the commercial market." This means that the needs and requirements of the

[1] The definition of legacy for this thesis will be as defined in [Ref. 1:p. 1] as any system that has been "designed, developed, and fielded."

military have had diminishing influence on the products that industry designs and produces.

The previously discussed budget shortfalls along with the reduction of influence in the commercial electronics sector have caused military avionics systems in general to fall further behind current technology. As these legacy avionics systems get older, the costs for modernization along with the costs to support the current systems continue to increase. Therefore, the need is clear for a way to modernize these aging systems that will lower these costs in the future.

B. POTENTIAL SOLUTIONS TO THE LEGACY PROBLEM

The solution to a problem as complex as the legacy avionics issue is not clear. The overall solution will lie in changes to design methods and acquisition policies that will continue to look for the benefits promised by COTS integration. Most importantly, the solution must also address the additional unforeseen problems that this integration has brought with it in a more far-reaching way.

This solution to the legacy avionics problem as a whole is too complex to be covered in one thesis. This thesis therefore will narrow the subject to address the area of microprocessors and their associated communication interfaces. This area can be considered of central importance to the problem as a whole. This is because microprocessors are so central to the performance of any avionics systems that any increase in performance of the processor will in turn almost guarantee an increase in performance of the entire system.

In his master's thesis, CDR Mike Croskrey [Ref. 1], investigates the possible solutions for the legacy avionics problem as they apply to microprocessors. He suggests several solutions to the problem and compares and contrasts the benefits and drawbacks of each. These solutions and their advantages are summarized in Table 1.

Proposed Solution	Advantages
Upgrade to a COTS binary compatible microprocessor, when available.	• Maintains old code and allows incremental updates using the new processor • Assures functionality of existing code
Maintain old processor or capability of executing the old code with hardware 1) Keep old processor board and add a COTS processor board 2) Develop a dual instruction set processor 3) Port the old processor to an ASIC 4) Port old processor to an FPGA	• Maintains old code and allows incremental updates using the new processor • Assures functionality of existing code • ASICs are fast and have low power requirements • FPGA relatively easy to modify if problems found
Maintain the capability of executing the old code using a software emulator	• Assures functionality of existing code
Port the old code to a new processor family	• May increase throughput
Translate the code to Higher Order Language (HOL)	• Improves ability to maintain knowledgeable workforce • Object oriented code facilitates reuse
Translate the code to COTS assembly language	• Facilitates use of a more current processor

Table 1. Solutions to Replacing Legacy Processors [From Ref. 3]

The solution that this thesis will focus on is the design and implementation of new hardware that is binary compatible with the existing processor and therefore able to execute the existing code. This hardware solution will also be binary compatible with all external interfaces since these components will not be redesigned as part of this thesis.

C. REENGINEERING

Forward engineering is the process of creating a new system and can be roughly broken down into three stages or processes. These stages include requirement specification, design, and implementation. The process of designing a system to replace an existing legacy system requires additional design steps in order to recover the design that is to be replaced. These additional steps can be grouped into a process called reverse engineering. Reverse engineering is the process of analyzing a subject that serves to identify its components and their interrelationship as well as produce a representation of

the system at a higher level of abstraction. Its primary purpose is to "increase the overall comprehensibility of the system for maintenance and future development." [Ref. 6, p16]

Reverse engineering can include the same steps defined in forward engineering but in reverse order. It also includes an additional step, or sub area, termed design recovery. Design recovery is a process in which domain knowledge, external information, and deduction are combined with observation to identify higher-level abstractions than those obtained directly. It is basically the process that combines all available resources to reproduce the information that allows a complete understanding of what the system does and how it does it. [Ref. 6]

In order to design and implement a new system that will replace an existing system, both the reverse and forward engineering processes must occur. This overall process, of both reverse and forward engineering, is termed reengineering. It can be defined as "the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form." [Ref. 6, p15]

The concept of rapid prototyping is a process that provides the means to produce prototypes of a design early in the design process. These prototypes allow the testing of key aspects of the design continuously throughout the design stage so the effects of early design decisions can be determined before other design decisions are made. The benefit of these prototypes increases as the complexity of the overall design increases.

In reengineering, rapid prototyping has an additional benefit that can both speed the design process and validate the design. This additional benefit is the ability to test the prototype using the environment and tools available to test the original design. This is especially important in complex designs or designs that lack detailed documentation.

D. PURPOSE OF STUDY

The purpose of this study is to investigate the process of reengineering a legacy avionics system, particularly the memory and communication interfaces of an embedded microprocessor system. It will include the implementation of the recovered design using Field Programmable Gate Array (FPGA) technology. It targets the AN/AYK-14(V) Navy

Standard Airborne Computer; specifically the XN-8 chassis used onboard the F-18 C/D aircraft. This computer was chosen not only because it is representative of the legacy avionics challenge already addressed, but also because the AYK-14 is the focus of an analysis of alternatives being conducted by the Naval Air Systems Command (NAVAIR) Advanced Weapons Laboratory (AWL).

The secondary purpose of the design recovery will be to serve as a reference for designers and programmers who are continuing work on the AN/AYK-14.

THIS PAGE INTENTIONALLY LEFT BLANK

II. DESIGN RECOVERY

A. OVERVIEW OF REENGINEERING PROCESS

Chapter I defined the terms that describe the process and the steps involved in engineering processes, which are illustrated in Figure 1.

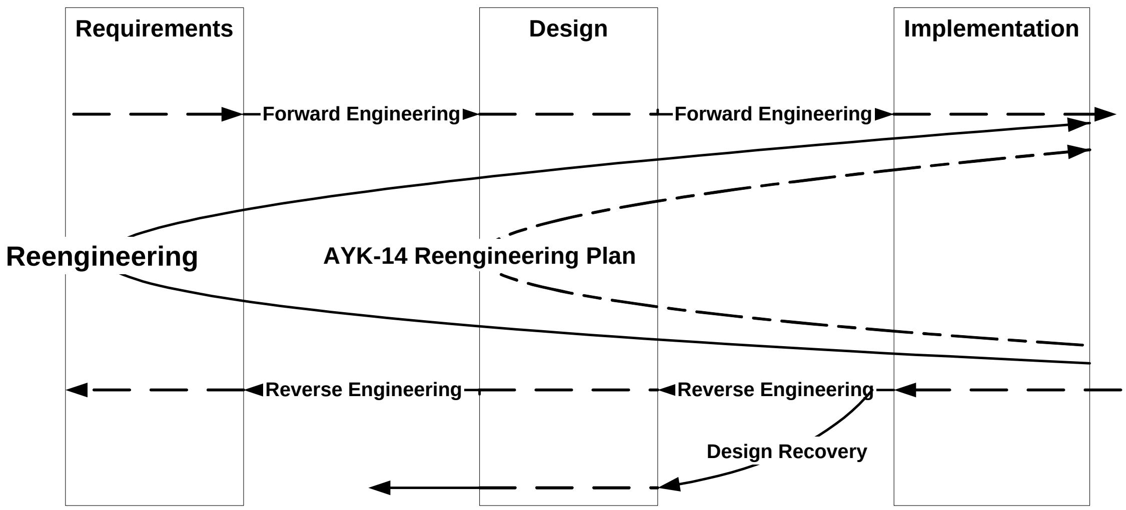


Figure 1. Engineering Processes

The AYK-14 Reengineering Plan, adapted from Reference 6 and shown in Figure 1, helps to depict the steps that were followed in this thesis. The key point that is illustrated is that the AYK-14 reverse engineering phase only investigated to the level of the design. The requirements were not analyzed directly for numerous reasons. First, the primary goal of this project was to design a replacement for the AYK-14 processor that was binary compatible with the rest of the system, therefore there was little room for changes to the overall design that would better meet the requirements. Another reason was simply that the time and resources available to continue the design recovery to the requirements level were not available. It should be pointed out that the requirements were researched at a high level as part of the design recovery to aid in the understanding of the design and implementation.

B. OVERVIEW OF THE AYK-14

An understanding of the mission and history of an avionics system is essential to the recovery of its design. This section will give a brief introduction to the AYK-14 to help define components and their roles. However, it is recommended that the reader refer to References 1 and 7 for a more detailed analysis and background on the system. The documentation supporting the AYK-14 was produced at varied times in the computer's lifecycle and therefore only considers equipment available at the time it was authored. This section is also intended to illustrate all of the major components of the system, even if they are outdated, in order to provide a reference when referring to the documentation. All of the documentation used in the design recovery is listed in Appendix A.

1. History of the AYK-14

Development of the AYK-14 began in 1976 by Control Data Corporation. It was designated the Navy Standard Airborne Computer in 1986. Since then, the AYK-14 has been used on seven types of Navy and Marine Corps aircraft including the AV-8B, F-14D, and F/A-18C/D. It consists of a family of modules that fit into a plug-compatible backplane. These modules can be broken down into four groups by function and they include processor, I/O, memory, and power. As the AYK-14's requirements have changed and technology has improved, the modules in each subsystem have evolved to increase overall capability. Therefore, there are numerous versions of the AYK-14 based on platform requirements and modules present.

2. Processor Subsystem

The processor in the AYK-14 has evolved through three generations of upgrades. The first generation is the central processor unit (CPU), which consists of three double-sided modules: general processor module (GPM), processor support module (PSM), and extended arithmetic unit (EAU). The second generation is the single card processor (SCP) that combines the three modules of the CPU into one module. The third generation processor is the very high-speed integrated circuit (VHSIC) processor module (VPM). An attribute of the VPM that is important to highlight is that it is the first processor to have onboard memory (1 M-word). The VPM is the processor that will be targeted for design recovery in this thesis.

There are two additional processors that are used solely for I/O functions. The first generation is the I/O processor (IOP), superceded by the extended I/O processor (EIOP).

3. Memory Subsystem

The memory subsystem consists of memory control modules and memory modules. The memory control modules provide access of the memory modules to the processor over the memory bus (MBUS or CPUBUS). There are three control modules: memory control module with memory (MCMM), memory subsystem module (MSSM), and the memory control module (MCM). There are four memory modules with four different forms of memory: DRAM memory module (DMM), programmable memory module, using EEPROM, (PMM), semiconductor memory module, using SRAM, (SMM), and core memory module (CMM).

4. Input / Output Subsystem

The I/O subsystem consists of a combination of I/O modules dependant upon the communication requirements. There are eight types of I/O modules that can be further classified as smart or standard. A smart I/O module has the ability to perform additional processing normally performed by the processor or I/O processor. This capability will be defined in greater detail in section H. The I/O modules interface with external equipment via buses or discretes. The I/O modules communicate with the processor via the I/O bus (IOBUS or XBUS). An AYK-14 can contain up to 16 I/O modules, with a maximum of five smart modules, depending on the Chassis used. The I/O modules and their classifications are listed in Figure 2.

5. Power Subsystem

The power subsystem is a single module that provides regulated power to all other systems. There are four types of module dependant upon the power requirements of the system. They are the power converter module PCM –1, PCM-2, PCM-3, and PCM-60.

6. Chassis Subsystem

The chassis subsystem is the housing used to contain all of the modules. There are nine standard chassis types to meet the size and connection requirements of the different

AYK-14 roles. The chassis contains a backplane into which each module is plugged to provide communication.

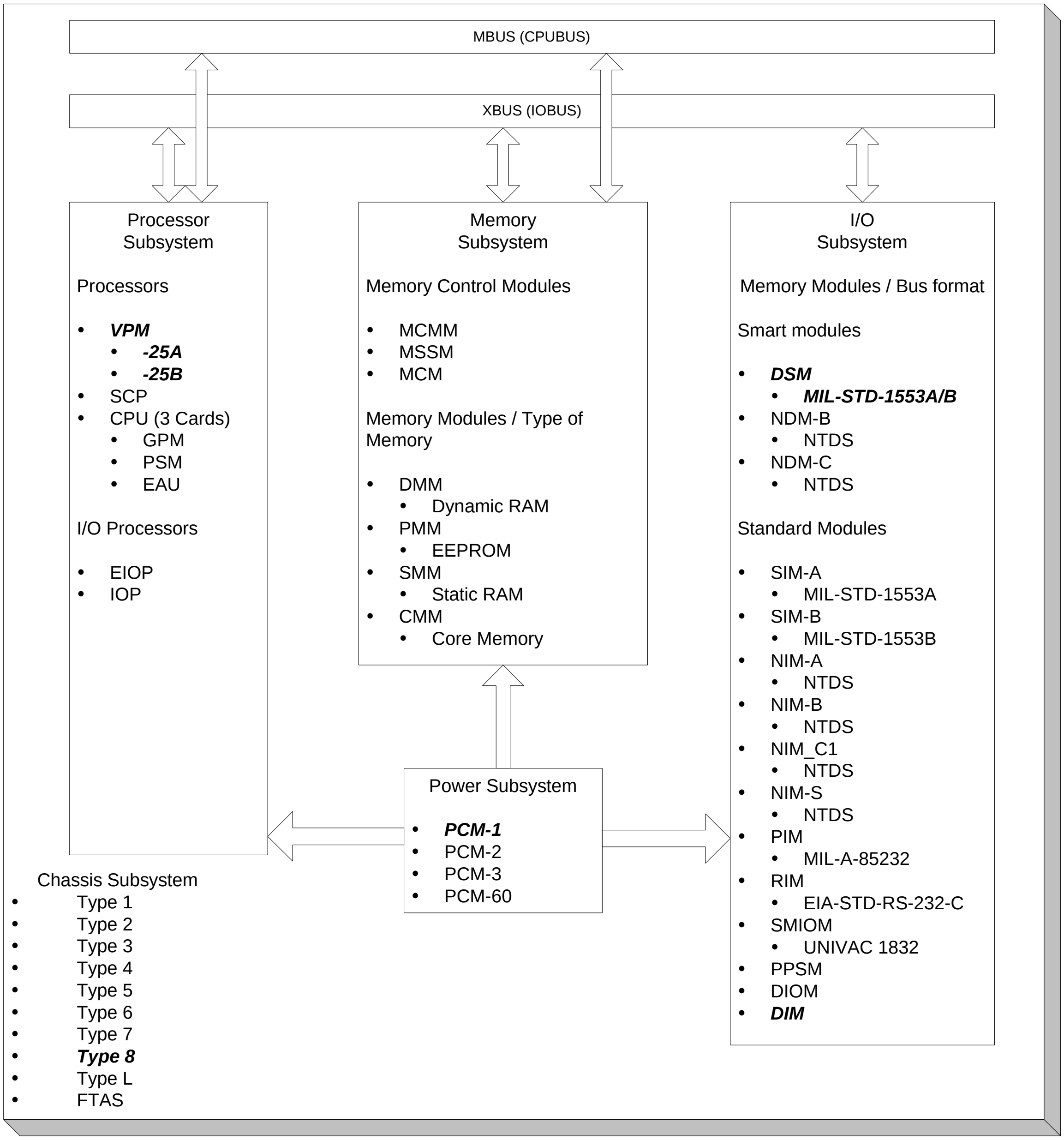


Figure 2. AYK-14 Subsystems

C. AYK-14 CONFIGURATION ON THE F-18C/D

The current AYK-14 configuration that is used on the F-18C/D is the CP-2360. It contains two VPMs (one 25B - Master, one 25A - Slave), six DSMs, one DIM, and one PCM-1 as shown in Figure 3. This is the configuration that was targeted for this thesis.

More specifically, the VPM processor as used in this configuration was the target of the reengineering process.

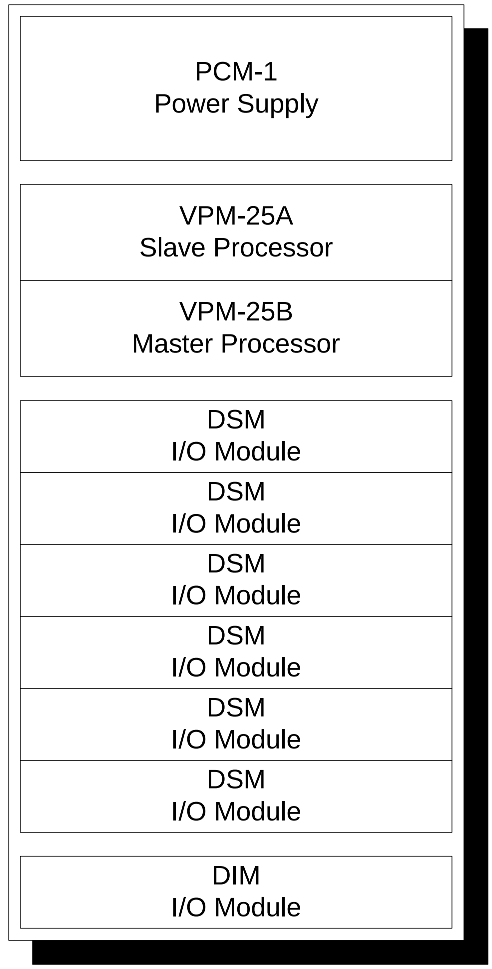


Figure 3. AYK-14 Chassis 8 – CP2360

The avionics system uses two CP2360's as Mission Computers, designated MC1 and MC2. MC1 processes all navigation and monitoring tasks and MC2 processes all sensor and weapons control tasks. The Mission Computers communicate with the other systems over six 1553 data-bus channels, as illustrated in Figure 4.Earlier F/A-18 aircraft use a chassis with only five 1553 data-bus channels.

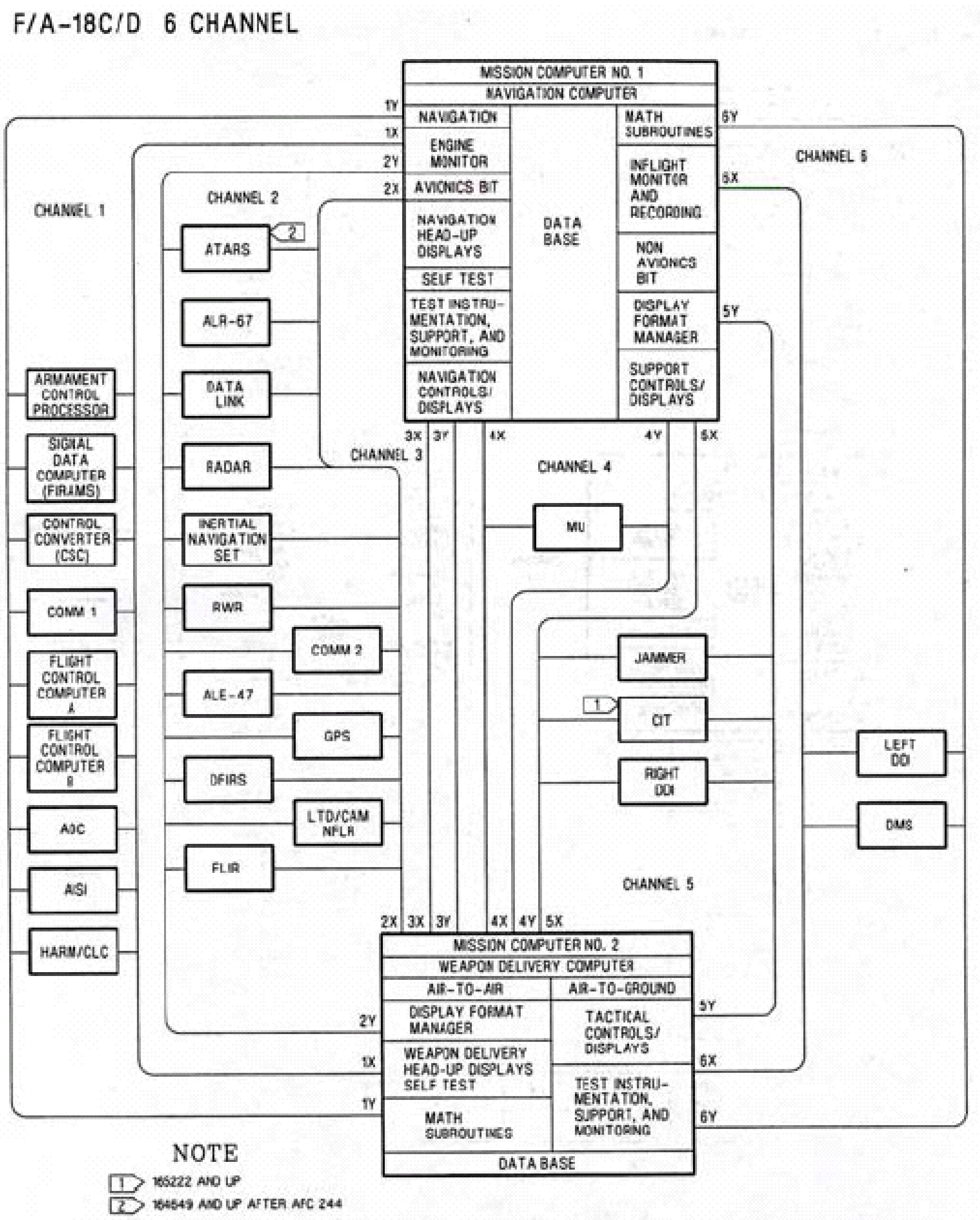


Figure 4. Six 1553 Data Bus Channels on F/A-18 C/D

D. VPM PROCESSOR

The VPM is a 16-bit Complex Instruction Set Computer (CISC) type processor with over 1 Million words of on-board memory. It is a 2-sided module that is organized into 3 major sections. These sections are the Instruction Execution Processor (IEP), Cache/Instruction Fetch (C/IF), and Adapter and are shown in Figure 5. The VPMs primary interfaces include the Input / Output Bus (XBUS or IOBUS), the memory bus

(MBUS), and the Event and Event Monitor busses (EBUS and EMON) along with multiple discretes. The A-side contains the 24 memory chips, the Adapter array, the MBUS and XBUS data and control signal buffers, and the external discrete receivers. The B-side contains four arrays, including the IEP and C/IF, 34 memory chips, and Event drivers and receivers.

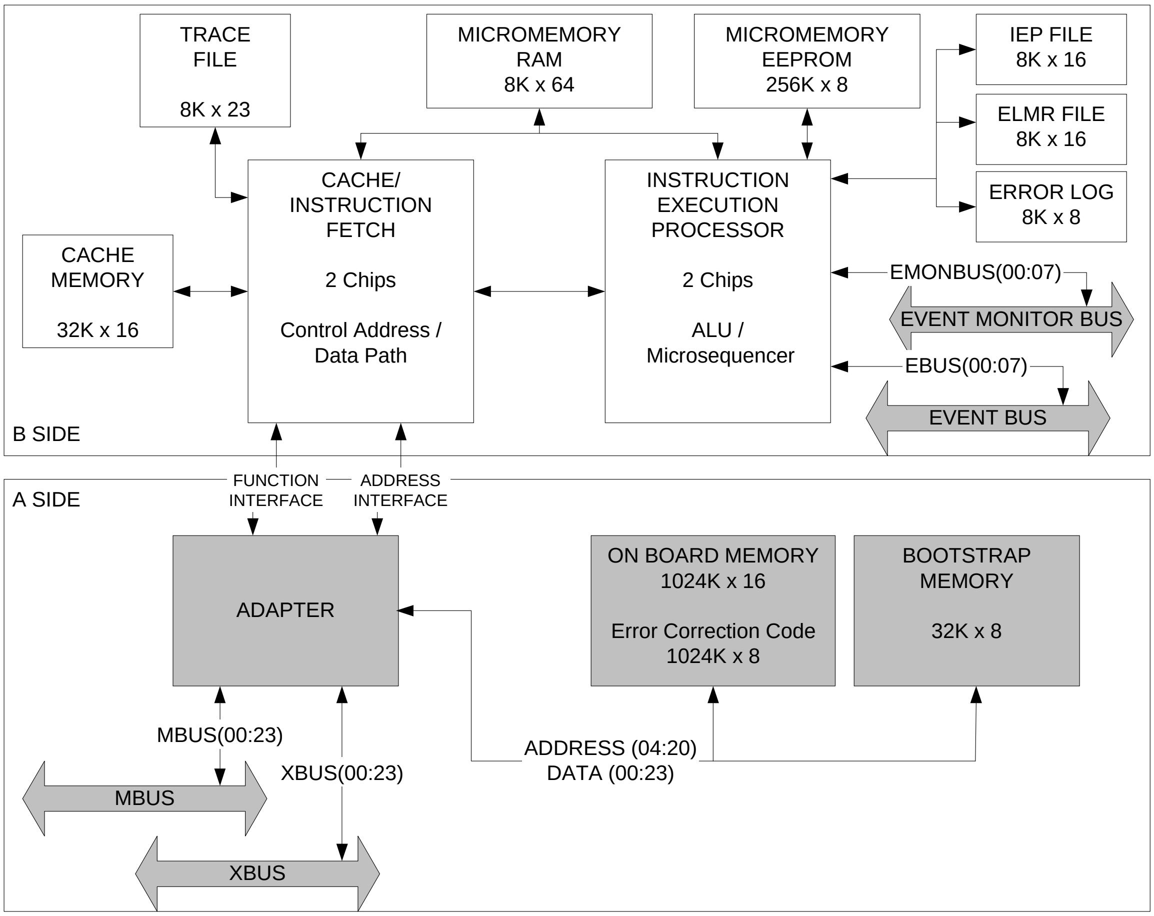


Figure 5. VPM Block Diagram

The IEP section is comprised of the microsequencer chip, the arithmetic chip, and the micromemory. It is implemented using a microprogrammed processor that executes microcode programs. Microcode programs control elementary parts of the processor and define the software instruction set used for the AYK computer. Every software command executable by the VPM is interpreted in the IEP by a series of microcommands. These commands, or microcode, are stored in EEPROM and downloaded to SRAM at start-up. The microcode stored in these memories is called firmware. Some other functions of

firmware include running BITs, servicing Events, and I/O operations. The IEP design was recovered and implemented by CDR M. Croskrey in his master's thesis and his design serves as the instruction processor for the design developed here. For additional details concerning the IEP design recovery, refer to Reference 1.

The C/IF section is comprised of the cache control and address chip, the cache memory, the data path chip, and the trace file. It provides the on-chip cache for the IEP and manages requests for memory to the adapter. The use of an on chip cache has been shown to significantly increase throughput and overall performance of most processors, however, the design recovery and implementation of this section is left to future students continuing work on this project due to time constraints.

E. ADAPTER

The primary function of the adapter is to control the onboard memory interface and the XBUS and MBUS interfaces. It handles all requests for memory from either the data path array, other VPMs via the MBUS, or I/O modules via the XBUS. It interfaces with the event system and contains two sets of page registers used for I/O memory references.

The VPM is capable of operating in two memory modes dependant upon the other modules present. These modes are standalone and non-standalone. In standalone mode, the VPM performs the role of Memory Controller and arbitrates memory requests and M and X bus usage. In non-standalone mode, a memory controller, such as the MCMM, is required to manage the memory. Both VPMs in the CP-2360 operate in the standalone mode.

The VPM is a 16-bit processor and the IEP and C/IF use 16 bits addresses for memory. The VPM has a memory reach of 8 million locations, which requires 23 bits for addressing. In order to reach this amount of memory, the VPM uses memory paging. The VPM uses banks of 64 16-bit wide Page Registers. The upper 6 bits of the 16-bit Software Address points to one of the 64 page registers. The contents of this register are used to create the complete 23-bit address, with 3 bits being used for memory protection. This 23-bit address is considered the absolute memory address and can address any location in the VPM memory range. The absolute address generation is depicted in Figure

6 for clarity. The control address array contains four sets of 64 page address registers used for generating the absolute address for on-board memory references.

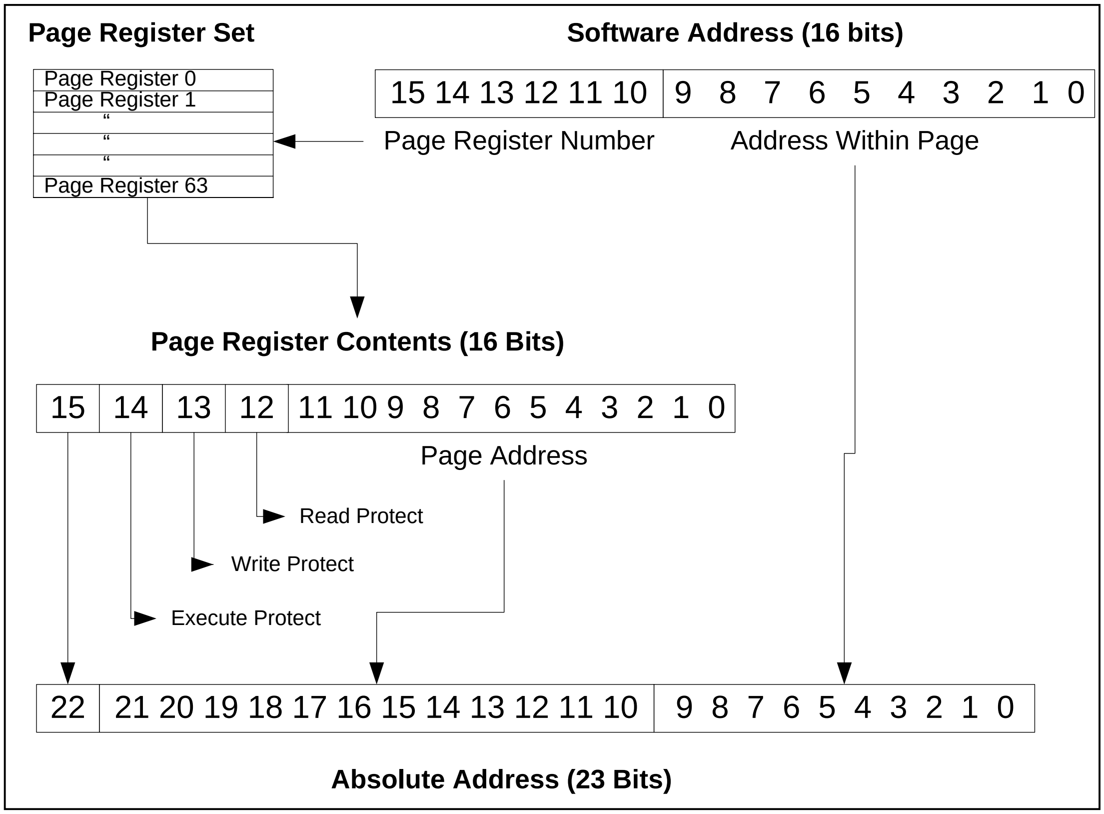


Figure 6. Address Generation

The VPM on board memory (OBM) consists of 1024K locations of 24 bit words. Each word contains 16 bits of data and 8 bits of error correction code. The memory is broken down into 256K of SRAM and 768K of EEPROM. The bootstrap memory consists of 32K addresses of 8-bit data organized as 16K of 16-bit word storage on a EEPROM. The lower 8K is loaded with bootloader programs for use on start-up or after a reset. The memory address range of the OBM is dependant upon the VPM’s location and role within the Chassis. The memory map of the entire address range is shown in Figure 7.

0 64K 1M 2M 3M 4M 5M

	MEM MOD or VPM-B	Master VPM	SLAVE VPM #1	SLAVE VPM #2	SLAVE VPM #3
	000000 00FFFF	EEPROM 100000 1BFFFF RAM 1C0000 1FFFFF	EEPROM 200000 2BFFFF RAM 2C0000 2FFFFF	EEPROM 300000 3BFFFF RAM 3C0000 3FFFFF	EEPROM 400000 4BFFFF RAM 4C0000 4FFFFF
PAGES	0 - 3F	400-6FF 700-7FF	800 - AFF B00 - BFF	C00 - EFF F00 - FFF	1000 - 12FF 1300 - 13FF

Figure 7. Absolute Address Assignment

F. EXTERNAL BUS OPERATION

The MBUS and XBUS (or IOBUS) are independent, 24-bit bi-directional busses that provide communication between the modules of the AYK-14. The MBUS is used to provide memory access to every VPM’s OBM and with memory modules. The XBUS is used for communications with I/O modules and for inter-processor communications (IPC).

The process of allowing modules to gain control of bus and transfer data on that bus is called bus arbitration. In standalone memory mode, the adapter of the Master VPM acts as the arbitrator for both busses. There are five primary control signals that are used for bus arbitration and control for each bus. These signals are DESIRE and GRANT for arbitration, and REQUEST, ACKNOWLEDGE, and RESUME for control.

Bus operations are initiated by the user and consist of two parallel word transfers. The first word is a 24-bit control word and is transferred from the VPM or smart I/O module to address a particular module and provide control information. The second word is a 16-bit data word that transfers data or status as input or output as determined by the function word.

1. Standalone Mode MBUS Operation

The VPM standalone mode of operation uses the memory control logic of the VPM that eliminates the need for a separate memory control module. Each VPM has access to the OBM of any other VPM, as well as memory modules if used. The MBUS functions as a 23-bit physical (post-paged) address memory bus, with the OBM address

allocation as shown in Figure 7. Each VPM performs its own paging and all I/O memory references use page set 0 on the master VPM. There is no interprocessor communication of page register or page state changes. Therefore, the paging and protection contained in each VPM is applicable only to that VPM. A single memory bus is used to prevent the interleaving of off-board memory references.

In standalone mode, the MBUS arbitration logic supports two external desire/grant signal pairs plus the processor's own desire/grant pair for a total of three users. Additional users can be added by daisy chaining the desire/grant signals. The version of the AYK-14 used in this thesis only has two MBUS users so the details of daisy chaining will not be covered here.

A user requests use of the bus by activating its DESIRE signal (active low). The desire signals of both external users are resynchronized before being used in the arbitration logic. The internal desire signal is captured in a flip-flop before it enters the arbitration logic. The synchronous desire signals are fed into the prioritization logic to determine which user is granted control of the bus. The algorithm makes use of a last user register that keeps track of which user was granted control of the bus last. The result is a rotating priority scheme based on which user had the bus last. The module that last used the bus drops to the lowest priority and the one following it gets the highest priority.

The arbitration algorithm outputs the next-user, which is fed into a latch that opens during the last half of the clock cycle. When enabled, the latch captures the next-user, which causes the appropriate GRANT signal to be enabled. The asynchronous and synchronous (post flip-flop) desire signals must both be active as a condition for activating a grant signal. This is to ensure that the grant is not activated before the desire signals are synchronized.

In addition to the five hand-shaking control signals, the VPM utilizes 10 additional signals for MBUS error detection and control. The signals are listed in Figure 8 and they include four parity bits, four control signals, a busy signal and an error signal. The first two control signals, MSB_WRITE and LSB_WRITE, indicate the type of memory operation, read or write. The other two control signals exist for future capability. The busy signal, M_BUSY, is used to indicate when the VPM is driving data on the bus.

The parity bits are used for error detection, with three used for the 24 address lines, for both the address and the data words, and one for the four command signals. The error control signal is used to indicate when a parity error is detected. The additional control signals are needed because all of the 24 bits are used for the address in the command word when operating in the standalone mode.

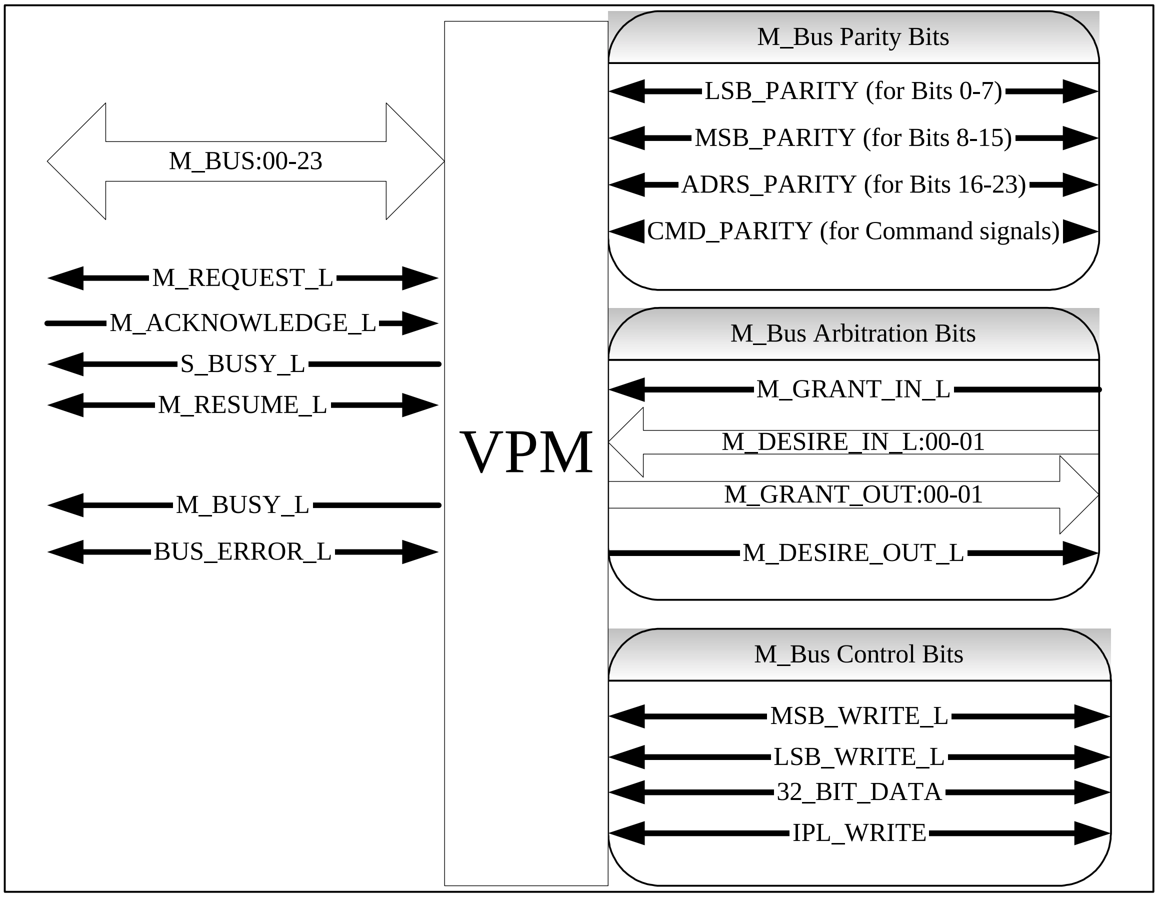
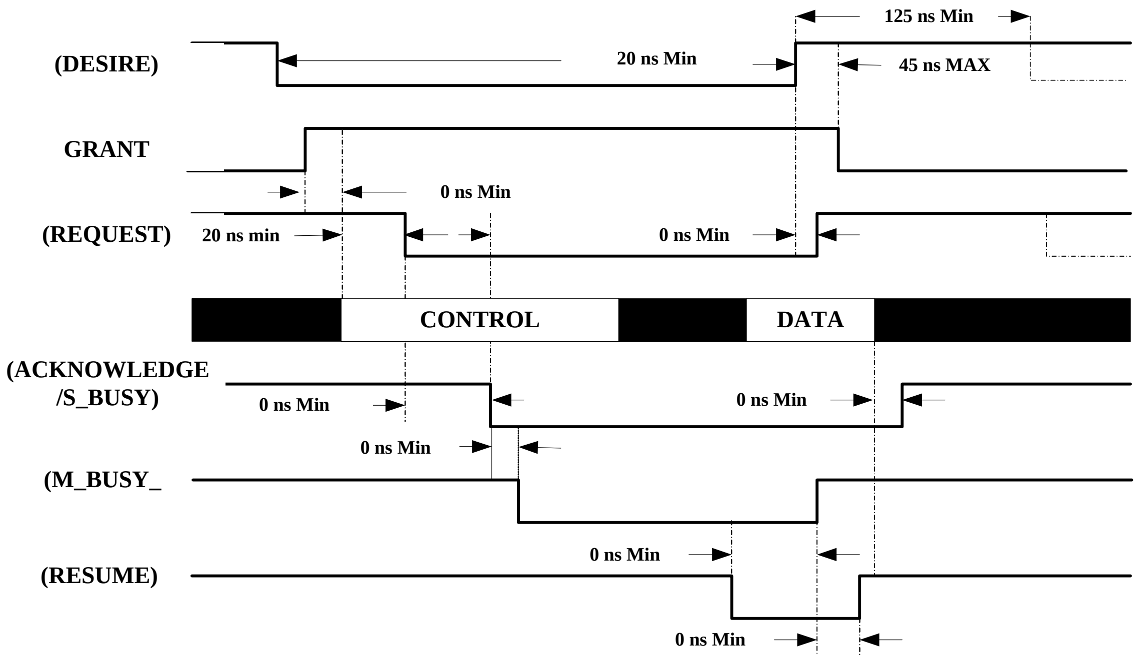


Figure 8. MBUS Interface Signals

After receiving control of the MBUS via a Grant signal, communication on the MBUS is initiated by the VPM activating a Request signal along with the 23-bit absolute memory address. The VPM also drives the four parity bits and the four additional control signals. The VPM who’s OBM is in the range of the address checks the parity of the address and the command signals. If there is an error, it activates the Error signal and stops responding to the memory request. If the parity check is successful, the responding VPM activates the Acknowledge signal and clocks-in the address. The initiating VPM activates its MBUSY signal to indicate that it is ready to either read or write data on the

MBUS. It will also deactivate the desire signal to the arbitration logic to allow the next user to be determined.

Output Operation



Input Operation

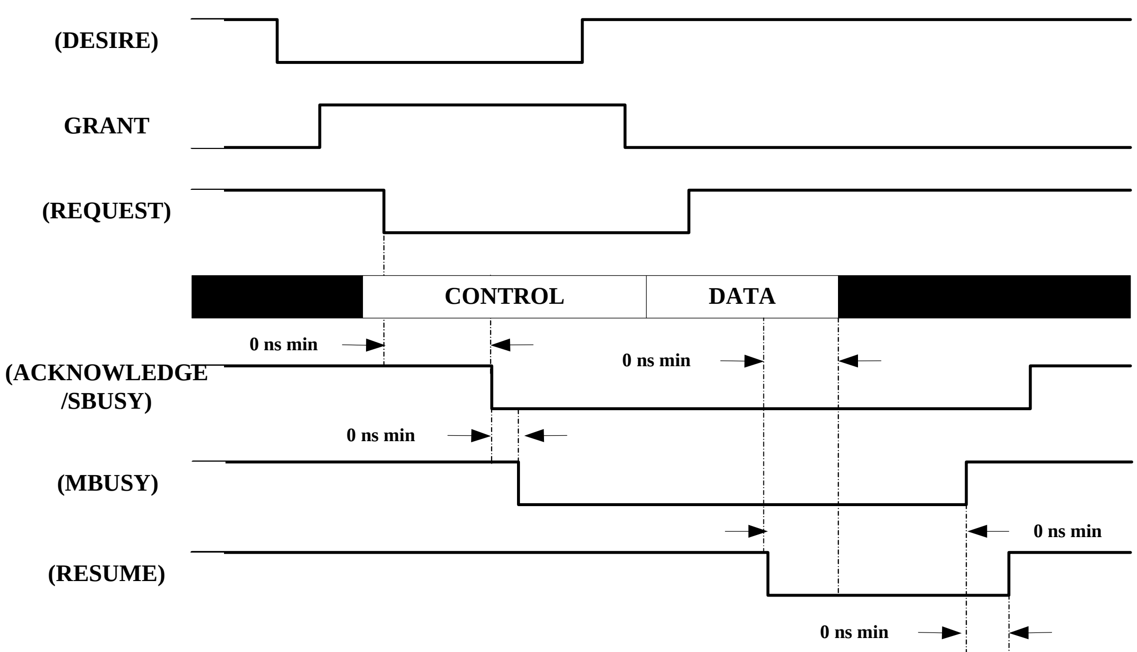


Figure 9. MBUS Standalone Operations

If the control signals indicated a read command, the initiating VPM will deactivate the Request signal and the responding VPM will drive the requested data on the MBUS along with the corresponding parity bits. When this data is valid, the responding VPM activates the Resume signal to indicate that the data is valid. The initiating VPM will clock-in the data and deactivate its MBUSY signal to indicate that the data has been read. The responding VPM will then stop driving the MBUS and deactivate the Resume signal to terminate the operation.

If the control signals indicated a write command, the initiating VPM will drive the requested data on the MBUS along with the corresponding parity bits and then deactivate the Request signal. When the responding VPM sees the deactivation of the Request signal, it clocks-in the data and activates the Resume signal. In response to the Resume signal, the initiating VPM removes data from the MBUS and stops driving the four control signals and the MBUSY signal. The input and output operations are illustrated in Figure 9.

2. Standalone XBUS Operation

The XBUS is the primary communication path between the processor and the I/O subsystems. All I/O control, instructions, and data transfer operations utilize this bus. For ‘smart’ I/O modules, the XBUS provides a means for direct access to OBM. The XBUS also provides an asynchronous channel for interprocessor communications. The XBUS interface signals are illustrated in Figure 10.

In standalone mode, the XBUS arbitration logic supports six external desire/grant signal pairs plus the processor’s own internal desire signals for a total of seven users. Additional users can be supported through daisy chaining of desire and grant signals. The Adapter on the master VPM monitors the external desire signals along with its own internal desire signal. The adapter arbitration logic determines the next user through a rotating equal priority process implemented in the same fashion as the MBUS arbitration previously discussed.

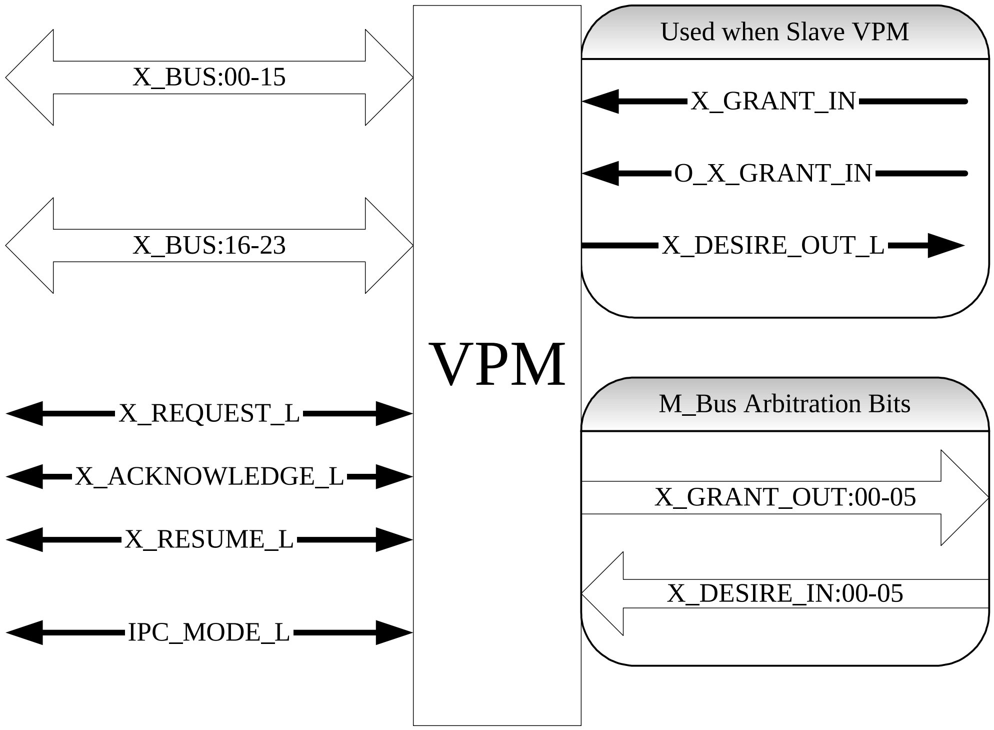


Figure 10. XBUS Interface Signals

The first step in XBUS communication is the Desire signal. Any module requesting use of the bus will activate its desire signal and wait for a response from the adapter. Once the adapter has determined the next user through the arbitration logic, it activates the Grant signal to that module. The owner of the bus then activates the Request signal while simultaneously driving the 24-bit control word onto the bus. The upper 8 bits of the control word, or XBUS Command Field, contain control information regarding the type of operation requested and the intended recipient. The lower 16-bits contain either a control word, an address, or data depending on the type of operation requested. Figure 11 illustrates the breakdown of the Command word and summarizes the meanings of the fields.

XBUS 24-Bit Command Word

Status Bit :

S :	= 1	Operation is a Status (input) or Function (output) operation
	= 0	Operation is a Data transaction

Output Bit :

O :	= 1	Transferring data FROM the initiating module TO the responding module (OUTPUT)
	= 0	Transferring data TO the initiating module FROM the responding module (INPUT)

Memory Bit :

M:	= 1	Operation is a memory type (memory read/write or memory status/function) CH# / CMD Bits are additional Command information
	= 0	Operation is an I/O operation CH# / CMD Bits are a Channel Number

CH#	Definition
00 - 0F	I/O Module Channel Numbers
10 - 16	Processor Channel Numbers
17	Broadcast to all Modules
1C	Broadcast to all IPC Modules
1D	Broadcast to all Smart I/O Modules
1E	Adapter Channel Number
1F	Broadcast to all I/O Modules

Figure 11. XBUS Command Word Format

After the module that was addressed decodes the control word, it activates the Acknowledge signal in response. If the operation commanded is an output, the module that issued the control word drives 16 bits of data onto the bus. The receiving module clocks-in the data and activates the Resume signal to indicate receipt. If the operation is an input, the commanded module activates the Resume signal, to indicate that it is now driving the bus, followed by driving the 16 bits of data onto the bus. The data will remain active for the duration of the Resume signal. Upon deactivation of the Resume signal, the arbitration logic will update the priority list and begin the process again. For I/O module broadcast operations, the Master VPM always generates the bus Acknowledge and Resume signals regardless of initiating module. For processor module broadcast

operations, the initiating module generates the bus Acknowledge and Resume signals. These steps are illustrated in Figure 12 for both input and output operations.

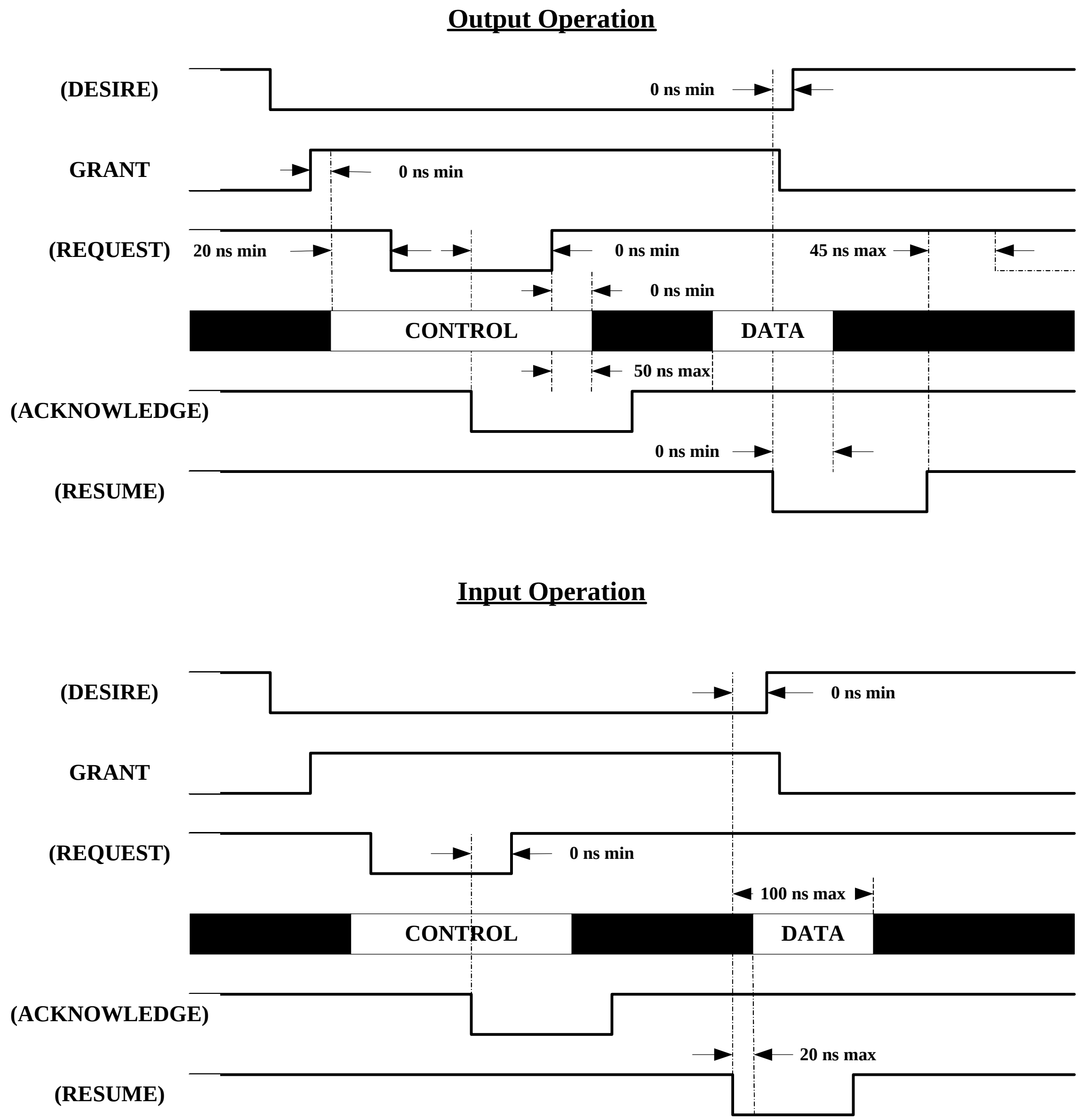


Figure 12. XBUS Timing Diagrams

When the XBUS is used for interprocessor communications, only bits 16-23 of the 24-bit bus are used for command and control along with the control and hand shaking signals. These 8 bits are referred to as the IPC BUS. Interprocessor communications consist of input and output transactions between VPMs and can be either from one VPM to another or broadcast to all VPMs in the system. The additional control signal used is

the IPC MODE signal and is connected to all VPMs. When activated, it causes all other VPMs to interpret Bits 16-23 as an IPC command.

G. EVENT SYSTEM

The event system is the mechanism by which the IEP is notified of conditions on the VPM, in other modules, or on other chassis that require servicing. It is controlled by the microsequencer array, part of the IEP, which monitors all sources for ‘active’ events. An active event is a condition or state that requires some type of action from the processor. Each event has a routine in firmware associated with it that can be called by the microsequencer to service the event.

The IEP, via firmware, checks for active events during idle loops when software is stopped or before each instruction is executed when software is running. The firmware interrogates for and handles all active events before it executes another software instruction. If more than one event is active, the microsequencer prioritizes the events based on a configuration dependant priority scheme. The event system provides a means of monitoring indicators, warnings, software chain execution, and external data transfers. There are two parallel subsystems in the event system; the polled event system and the direct event system.

The VPM also has an interrupt system similar to other processors in addition to the event system. Normal software execution is stopped for the handling of these interrupts. All of these software interrupts[2], not automatically trapped by microcode, are signaled via activation of associated events. The interrupts to the VPM can come from any module in the Chassis and are divided into three classes based upon their source. Class I interrupts deal with hardware failures or functions. Class II interrupts indicate software failures or functions. And Class III interrupts are for I/O failures or functions. The interrupts can be locked out by class, via software commands, by setting bits 12 – 14 in status register 1. All interrupts and the events associated with them are listed in Figure 13.

[2] The AYK-14 documentation refers to all three classes of processor interrupts as ‘software interupts’ because they can interrupt normal execution of the software for handling.

Class		Interrupt	Event Class	Event Discrete
Hardware	I	Power Fault	0	0/1
		Memory Timeout	5	1
		Memory Parity	5	2
		Hardware Fault Warning	5	3
		I/O Failure	-	-
		Thermal Overload	0	2/3
		Hardware Fault	5	6
Software	II	CP Instruction Fault	-	-
		I/O Instruction Fault	-	-
		Floating Point Under/Overflow	-	-
		Executive Return	-	-
		Executive Mode Fault	-	-
		Memory Protect Fault	6	0
		RTC Overflow	6	1
		Monitor Clock Overflow	6	2
		System Reset	6	4
		Processor Interrupt 0	6	6
		Processor Interrupt 1	6	7
		Fixed Point Overflow	-	-
		Module Overtemp	5	7
		External Interrupt 2	3	6
		External Interrupt 3	5	4
I/O	III	I/O Channel Abnormal Interrupt (ERI)	7	0/4
		External Interrupt (EII)	7	1/5
		Output Chain Interrupt (OCI)	7	2/6
		Input Chain Interrupt (ICI)	7	3/7

Figure 13. Software Execution Interrupts

1. Polled Event System

Polled events are events that occur on other modules that require servicing by the VPM processor. They deal primarily with software chain execution or external data transfers. They are referred to as polled events because the event monitoring system uses a polling sequence to determine which events are active. The event polling system consists of two 8-bit busses, the event monitor bus (EMON) and the event bus (EBUS). The EMON bus is driven by the VPM hardware and used to pass commands to manage the polling sequence. The EBUS is an open collector bus that is driven by the modules of the event system in response to commands on the EMON bus.

Polled events are organized by four attributes including priority, class, group, and discrete. Every event is assigned to one of three priority levels, and one of eight classes. An important note is that the event attribute of class is separate from the interrupt attribute of class. As an example, all class III interrupts shown in Figure 13 are listed in the event class seven. The binary form of the class, group, and discrete information of an event is used to form an event vector. This vector is used to point to the starting address in microcode of the event handling routine and is shown in Figure 14.

There are eight different classes of events, with four dedicated to I/O events and four to non-I/O events. The I/O events are further broken down into groups or channel pairs. Since there are only eight EBUS lines, the I/O modules must be grouped into the channel pairs to provide the ability for up to 16 I/O modules to activate events. This is explained in more detail when the polling sequence is covered. Within each class of events, there are eight discrete events for non-I/O events, and four for I/O events. All of the events are listed by class and discrete in Appendix B (See Microcode Reference Manual – p 4-17).

The event monitor continually queries the modules in the event system for events that have become active. It does this by cycling through a series states during which it determines which events are active, and which active event has the highest priority. These states are sent to the modules via the EMON bus and the modules responses are returned via the EBUS. The polling sequence is required because the modules on the EBUS do not each have discrete signals to indicate the presence of an event. The EMON bus is shown in Figure 14 along with a listing of the bits meanings.

a. 1st State: ESTATE = 01

The first state in the polling sequence is ESTATE = 01. In this state, the event monitor is requesting any active events from any module capable of initiating a polled event. When any module detects this state on the EMON bus and has an active event, that module will drive the EBUS line corresponding to the class of event that is active. If there are no active events, the event monitor remains in this state. If an event is detected on the EBUS, the event monitor will determine the highest priority class of event that is active and drive the ECLASS lines with that class value. If that class is an

I/O class (Class = 1,2,4,7), the event monitor will then transition to ESTATE = 10. If it is a non-I/O class (Class = 0,3,5,6) the event monitor will proceed to ESTATE = 11.

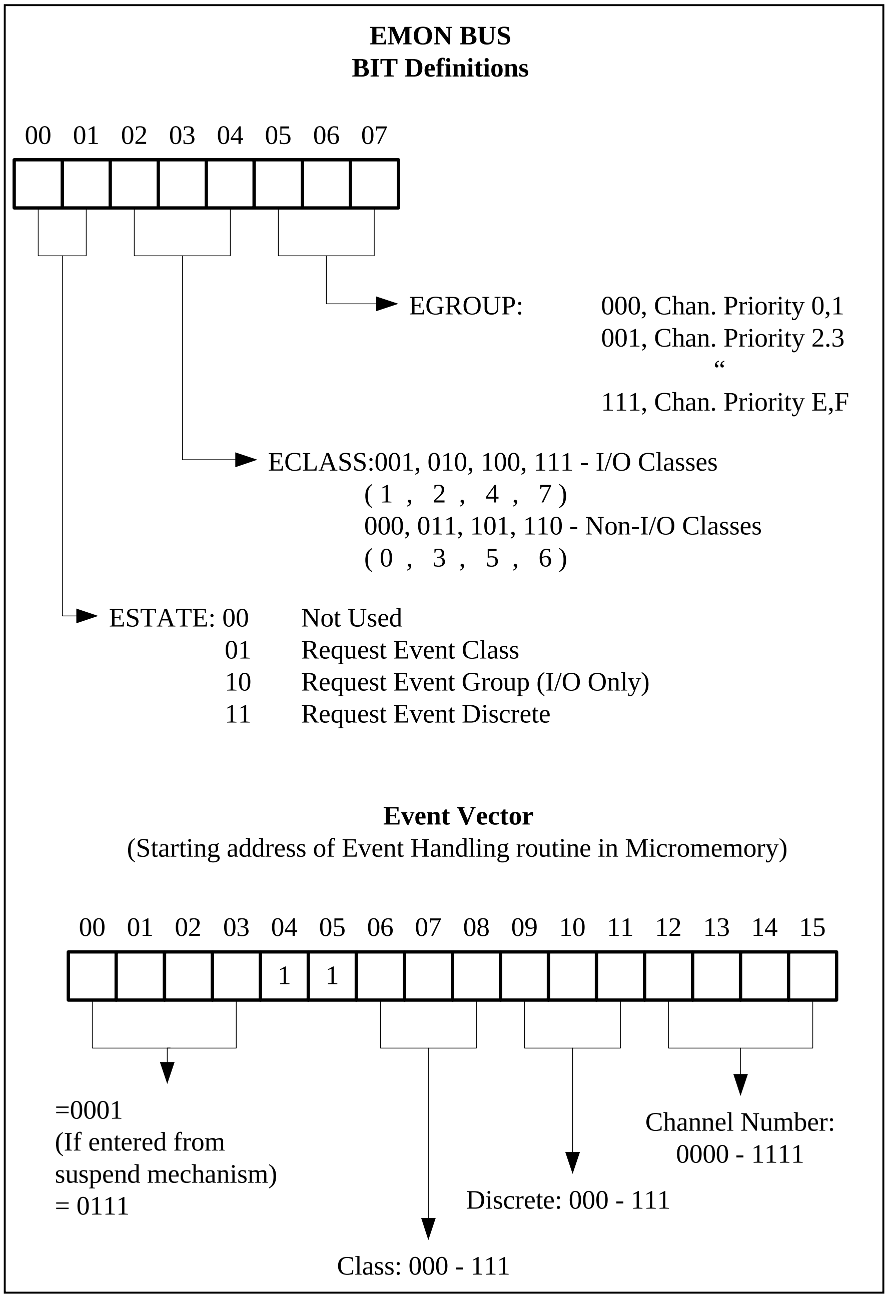


Figure 14. Event Monitor Bus Definition

b. 2nd State: ESTATE = 10

If the highest priority event class with an active event is an I/O class, then the event monitor will enter ESTATE 10. Along with the ESTATE bits, the monitor now

drives the ECLASS bits with the highest priority class with an active event. In this state, the event monitor is requesting all modules with active events in the class output on the ECLASS lines to respond on the EBUS lines. There are two I/O modules, or pairs, assigned to each discrete line. The event monitor will determine the highest priority channel pair based on the EBUS response and drive the EGROUP lines of the EMON bus with that value. The priority scheme used is a function of the wiring of the interconnect assembly for the assigned slot in the chassis. The event monitor will then transition to ESTATE 11.

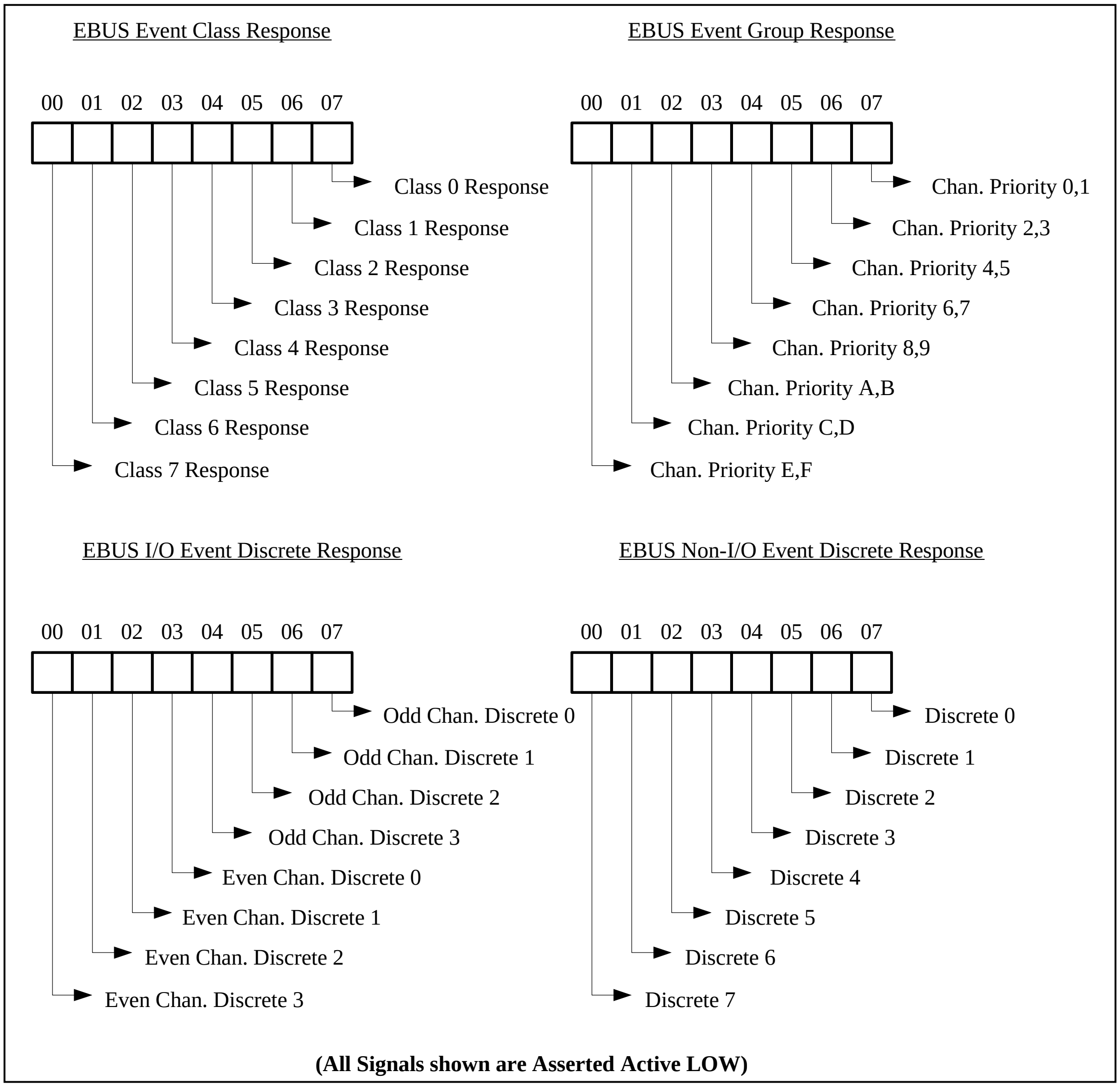


Figure 15. Event Bus Response Matrix

c. 3rd State: ESTATE = 11

If the highest priority event class with an active event is a non-I/O class, then the event monitor will enter ESTATE 11 directly from ESTATE 01. Along with the ESTATE bits, the monitor now drives the ECLASS bits with the highest priority class that has an active event. For an I/O class, the monitor will drive the highest priority channel pair, based on the determination from ESTATE 10, onto the EGROUP lines. For a non-I/O class, the monitor will drive the EGROUP lines to a known value corresponding to the class.

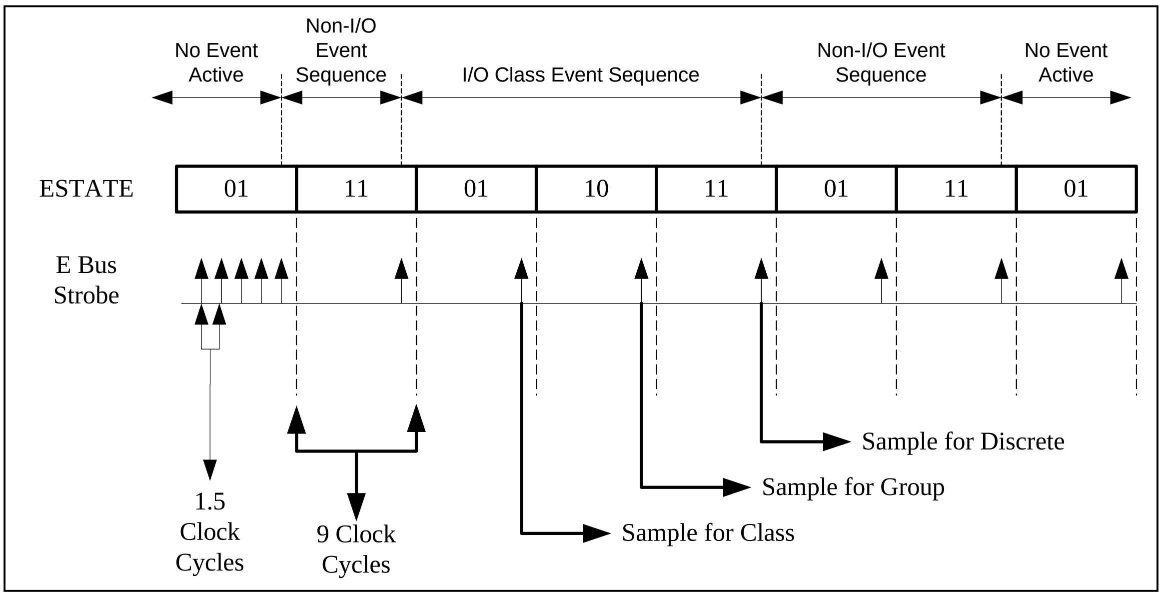


Figure 16. Event Monitor State Sequence

In this state, the module or module pair with the highest priority should now be the only one responding on the EBUS. For a non-I/O class, the responding module will drive the EBUS lines corresponding to the discrete events that it has active. For I/O modules, the EVEN module of the selected channel pair will respond on the lower four lines of the EBUS, and the ODD module will respond on the upper four lines. This restricts the I/O modules to only four events in each class.

The Class, Group, and Discrete values that are obtained are then used by the event monitor to generate the event vector, shown in Figure 14, for microcode handling of the highest priority event. After creating the event vector, the event monitor transitions back to ESTATE 01 and begins the sequence again. The EBUS responses to

each ESTATE is shown in Figure 15 and the timing for the polling process is shown in Figure 16 for additional clarity.

2. Direct Events

Direct events are generated in the control address, data path, and adapter arrays and sent to the microsequencer array. There are also direct events that come from off the module as well as some generated internally in the microsequencer array. There are 63 events that can be stored for handling in the direct event register. Direct events provide a means of notifying the event monitor of an immediate request for service from the firmware. It is more direct than the polled events but the events are still subject to priority logic and can be masked as well.

Direct events from the direct event register and the events generated in the polling sequence are filtered through a class mask. This mask is controlled via firmware and provides a means to stop specific classes of events from being seen by the priority logic. The priority logic compares all unmasked events and determines the highest priority event, which is then serviced by the firmware.

H. INPUT / OUTPUT MODULE OPERATION

The I/O modules provide the communication link between the VPM processors and other equipment in the system. The VPM communicates with the I/O modules via the XBUS and Event bus. The I/O modules communicate with other equipment via discrete signals and buses, specifically the MIL-STD-1553 data bus for the configuration recovered. The I/O modules are categorized as smart or standard based upon the amount of on-board processing they are capable of executing.

1. I/O Channel Software

There are three types of commands that are used to control the I/O modules operation. The first two types are ‘user’ commands that are used in operational programs and are considered software commands. Some of the capabilities provided are the ability to initiate and halt I/O channel operation, enable and disable I/O channel interrupts, load and store control memory words, and read I/O channel status.

The first type of command controls the initiation of all I/O channel operation. This command is the Input / Output Command Request (IOCR), Op Code 7400. This processor instruction, when encountered in the software during normal program execution, causes the processor to execute the instructions at a specific location in main memory called the command cell. The location of the command cell is 0060 and 0061 if the executing VPM is operating as the master, and 0062 and 0063 if it is operating as the slave. The IOCR is used in the main source code to start or stop I/O channel programs, monitor or modify channel operations, and modify Control Memory locations.

The second type of command is the set of processor executable commands that are used in the source code to control I/O operations. These commands can be broken down into three classes, including Command Instructions, Chain Instructions, and Command/Chain Instructions, and are listed in Appendix C. The Op Codes for these commands fall in the range E0-FF and are illegal unless executed following an IOCR command. These commands can be executed by the VPM or by a Smart I/O modules. These are the commands that are used in the programming of I/O channel functions.

The third type of command is the set of command words that can be sent as the control word of an XBUS operation. These commands are generated by the adapter and are used to either pass processor executed commands to the I/O module for additional action or to command I/O module action in response to an active event. These commands can be either broadcast or addressed to an individual module and can be either two word (command word and data word) or one word (command word only, data word is ignored) commands. All of these adapter generated commands are listed in Appendix D (Table A-2 and A-3 from design guide for I/O modules).

2. I/O Channel Control Memory

Each I/O channel has associated with it a 16-bit by 16-word control memory. This memory is located on the VPM for standard I/O modules, but is located on the I/O module for Smart I/O modules. The format and definition of each word in a control memory is dependant upon the module, however, most modules contain the same basic words. The control memory contains parameters that are used in the operation of the associated I/O module, such as pointers to programs, word counts, and status words. As

an example, the Control Memory for the DSM is listed in Figure 17 with a brief explanation of each word's function.

Location	Control Word	Description
0	Spare	
1	Spare	
2	Spare	
3	Bit Jump Word (BJW)	Used with bit jump Chain Instruction
4	Spare	
5	Buffer Address Pointer (BAP)	Address of the next memory location in the data buffer
6	Chain Address Pointer (CAP)	Address of the next Chain Instruction to be executed
7	Address Table Pointer (ATP)	Used to calculate BAP as part of data transfer command
8	Command Word 1 / Status Word 1	Contains word used in 1553 protocol (depending on mode)
9	Command Word 2 / Status Word 2	Contains word used in 1553 protocol (depending on mode)
A	Message Control Word 1 (MCW1)	Personality dependant mode and control information
B	Message Control Word 2 (MCW2)	Control information common to all personalities
C	Discrete Control Word (DCW)	Control info which selects mode of operation for discretes
D	Discrete Input/Output Word (DIOW)	Used for masking of discretes
E	Interrupt Clear Word (ICW)	Used in association with the Discrete Interrupt
F	Chain Table Pointer (CTP)	Used to support Tabular Output Operations

Figure 17. DSM Control Memory

3. I/O Channel Chain Programs

All I/O channel operation is initiated through the execution of the IOCR instruction by the processor. This instruction causes the processor to process the instruction in the command cell (memory locations 0060-61 or 0062-63). The instruction in the command cell will be an instruction that initiates activity on one of the I/O channels. There are two forms of I/O channel activity; I/O information transfer and I/O program execution or Chaining.

A chain program is a set of instruction, located in main memory, which perform an operation on an I/O channel. The program is made up only of chain instructions that are listed in Appendix C. The program normally transfers parameters between main memory and the I/O channel Control Memory, and initiates transfer of blocks or buffers of data or control words on the channel interface lines. Multiple I/O channels can have I/O chains active concurrently, with the event system providing regulation.

An important concept to emphasize is the difference in how chain programs are executed in standard and smart I/O modules. Standard I/O modules do not have the capability to execute software instructions (the first 2 types of commands previously discussed). Their chain programs are executed through the VPM processor executing the software commands in the chain program and sending corresponding commands (the third type of command previously discussed) over the XBUS to command the I/O module. The VPM time shares the execution of chain commands between the operational program and among the I/O modules with active chaining.

Smart I/O modules are capable of executing directly all of the software instructions that can be used in chain programs (i.e. all commands from Appendix C.) This means that once an I/O operation is initiated via an IOCR command, the VPM will continue processing the operational program and the smart I/O module will execute the chain program. It is able to do this by accessing the chain instruction directly from memory using the XBUS.

4. I/O Channel Software Interrupts

Class III software level interrupts are associated with I/O module operation. These interrupts can be enabled or locked out on an individual channel or as a group. They are handled via an interrupt handling routine that the processor is vectored to upon interrupt recognition. These interrupts are listed in Table 2.

Class	Priority	Interrupt	Definition
III	1	ERI	Error Interrupt
III	2	EII	External Interrupt
III	3	OCI	Output Chain Interrupt
III	4	ICI	Input Chain Interrupt

Table 2. I/O Channel Interrupts

ERI interrupts are generated upon detection of an error condition. EII interrupts are generated when the I/O module receives a channel interrupt word. The interrupt word is stored in a table in main memory prior to generation of the interrupt. The address in the table is 80 plus the channel number (80-8F). OCI and ICI interrupts are generated when

the chain program on the associated channel encounters and executes the Interrupt Processor (IPR) instruction.

5. I/O Channel Events

There are four classes of events that can be set by I/O modules to signal active events to the VPM. These events are used to communicate the progress of data transfer operations and chain programs, and to signal software interrupts. All of the I/O events are listed by class and discrete in Figure 18 and a description of each is given in Table 3.

The event system provides a means for the processor to efficiently manage the numerous operations occurring on the I/O channels. It allows the processor to start an operation on an I/O channel and then to continue executing the executive code while the I/O channel performs its tasks. The events allow the I/O cannels to notify the processor when it has completed a task and either needs more information or is ready for another task. It is a means of providing parallel operation of all the I/O channels.

For example, when an I/O chain program is in progress on a channel, that channel will raise the Input or Output Chain Request Event. While this event is active, the processor will continue to execute instructions in the corresponding chain program. When the VPM executes an instruction that indicates a chain program is complete, the firmware will notify the I/O module via an XBUS command. The I/O module will then deactivate the chain event.

I / O Class Events

Event Class	Name	Event Bus Discrete							
		Even Channel				Odd Channel			
		Name	Name	Name	Name	Name	Name	Name	Name
1 (001)	Indexed Data Transfer	Remote Terminal Command	Output Data Request 1	Input Data Request 1		RTC	ODR1	IDR1	
2 (010)	Data Transfer	Unique Channel Request	External Interrupt Request 2	Output Data Request 2	Input Data Request 2	UCR	EIR2	ODR2	IDR2
4 (100)	I/O Chain	Map	Output Chain Request	Input Chain Request	External Interrupt Request 4	MAP	OCR	ICR	EIR4
7 (111)	Class III Interrupts	I/O Channel Abnormal	External Interrupt	Output Chain Interrupt	Input Chain Interrupt	ERI	EII	OCI	ICI

Figure 18. Input / Output Channel Events

In Figure 18 it should be noted that the Even and Odd channels have the same events, however, the Acronyms for the events are listed for the Odd channel to provide a reference. Also, the repeated discrete events (i.e. ODR1, ODR2) provide for a hierarchy of event priorities.

6. I/O Channel Basic Operation

The operation of either standard or Smart I/O modules involve communication on the Event bus, XBUS, and possibly the MBUS. Multiple I/O channels can be operating chain programs or data transfers at the same time with the event system and priority logic providing deconfliction and minimizing the amount of time that the processor spends waiting for a response from the I/O module.

I/O Module Event Descriptions

Class 1: Indexed Data Transfer		
Remote Terminal Command	RTC	Causes the Processor to request an Index Status Word from the I/O Module via the XBUS. The status word is used with the Address Table Pointer (CM-7) to generate a new output Buffer Address Pointer (CM-5)
Output Data Request 1	ODR1	Causes the Processor to send a data word to the I/O module as determined by the BAP. This is the highest priority ODR and is used to give priority to time-critical I/O modules.
Input Data Request 1	IDR1	Causes the Processor to request a data word from the I/O module and place it in main memory at the location pointed to by the BAP. This is the highest priority IDR and is used to give priority to time-critical I/O modules.
Class 2: Data Transfer		
Unique Channel Request	UCR	Causes the Processor to request a unique function word from the I/O module. Depending upon the function code returned, the processor will perform a given function. This is used if I/O module needs additional capability.
External Interrupt Request 2	EIR2	Causes the Processor to request an interrupt word from the I/O module. This event is implemented in conjunction with the Class 7 EII event to provide the instruction that is processed in the interrupt. This event is a higher priority event than EIR4.
Output Data Request 2	ODR2	Causes the Processor to send a data word to the I/O module as determined by the BAP. This is the lower priority ODR.
Input Data Request 2	IDR2	Causes the Processor to request a data word from the I/O module and place it in main memory at the location pointed to by the BAP. This is the lower priority IDR.
Class 4: I/O Chain		
Map	MAP	Causes the Processor to request a status word 0 from the I/O module. The status word provides the modules channel number and type code. This information is used to construct a MAP table of all I/O modules in the system.
Output Chain Request	OCR	This event requests the processor to execute the next output chain instruction located at the address pointed to by the output chain address pointer .
Input Chain Request	ICR	This event requests the processor to execute the next input chain instruction located at the address pointed to by the input chain address pointer.
External Interrupt Request 4	EIR4	Causes the Processor to request an interrupt word from the I/O module. This event is implemented in conjunction with the Class 7 EII event to provide the instruction that is processed in the interrupt. This event is the lower priority EIR
Class 7: Class III Interrupts		
I/O Channel Abnormal	ERI	Causes the Processor to generate a class III, priority 1 software interrupt. Used as an error reporting mechanism by the I/O module.
External Interrupt	EII	Causes the Processor to generate a class III, priority 2 software interrupt. Used in conjunction with the EIR event. This is the lowest priority class of event so that the higher class EIR can load the memory with the interrupt information first.
Output Chain Interrupt	OCI	Causes the Processor to generate a class III, priority 3 software interrupt. Used to notify processor when a certain point is reached in a chain program. For example, if the I/O module is ready to begin data transfer.
Input Chain Interrupt	ICI	Causes the Processor to generate a class III, priority 4 software interrupt. Used to notify processor when a certain point is reached in a chain program. For example, if the I/O module is ready to begin data transfer.

Table 3. I/O Event Descriptions

I. DISCRETE AND SERIAL MODULE

The Discrete and Serial Module (DSM) is a Smart Input / Output module that provides the AYK-14 with two interfaces to external equipment. One interface is a serial multiplex input/output interface in accordance with MIL-STD-1553A/B. The other is a 16-bit input/output/discrete interface. The DSM is considered a ‘Smart’ I/O module because it has the capability to execute chain instructions, to read and write directly to memory, and to control the 1553 interface. All of the DSM’s interfaces are illustrated in Figure 19.

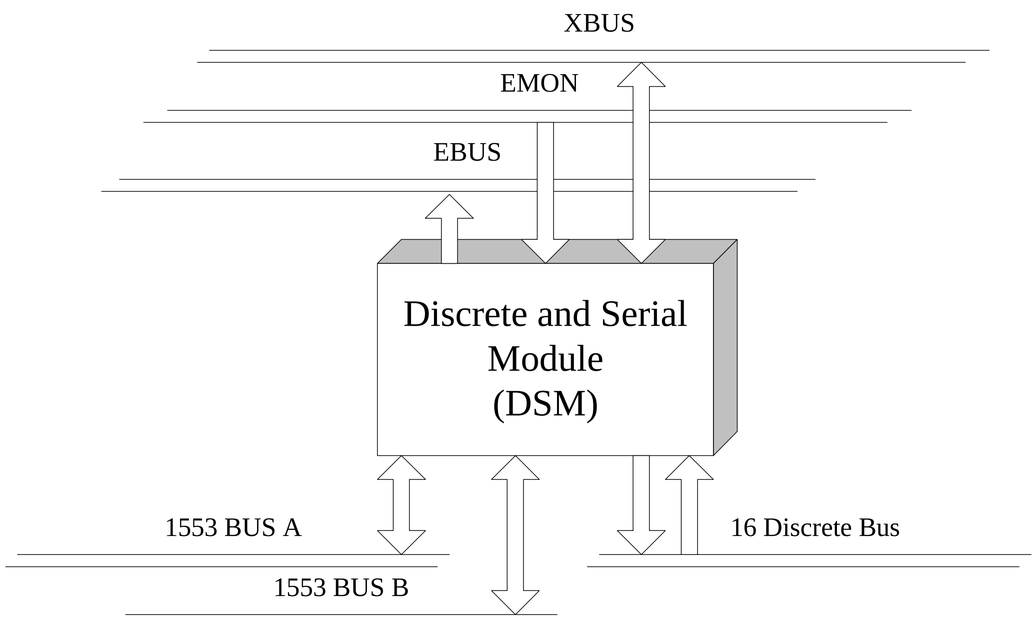


Figure 19. Discrete and Serial Module Interfaces

1. DSM Personalities and Modes

The DSM can be configured to operate in different configurations in order to provide flexibility and adaptability to the AYK-14. These configurations allow the 1553 portion of the DSM to perform like earlier I/O modules, specifically the SIM-A and SIM-B. The DSM can be configured with three personalities that include the SIM-A, SIM-B, and Alternate SIM-B. The SIM-A personality provides the capability to operate using the 1553A protocol. The SIM-B personality provides both the 1553A and 1553B protocol. Finally, the alternate SIM-B adds additional restrictions concerning chaining operation in addition to the 1553A/B capability. In every personality, the 1553 interface of the DSM

can operate in one of three modes, which include Self-test, Remote Terminal/Bus Monitor, and Bus Controller. These modes define the role of the DSM within the 1553 bus architecture.

2. Smart I/O Operation

The two features of the DSM that distinguish it from other I/O modules and make it a ‘Smart’ module are first, the ability to read and write directly to memory, and second, the ability to execute I/O instructions. This capability provides a good deal of autonomy to the DSM and greatly reduces the number of instructions that the VPM is required to execute during any I/O operation. The DSM has the ability to execute most of the I/O command and chain instructions in the VPM’s instruction set.

The initiation of operations on the DSM still requires the VPM to execute an IOCR instruction. Once initiated, the DSM requests the command or chain instructions directly from memory via an XBUS operation using its on-board Control Memory. The on-board control memory is an important distinction between standard and smart I/O modules. The presence of the information contained in the Control Memory on-board is essential for the DSM to request and execute it’s own instructions. For example, in order for the DSM to request a chain program instruction, it must have the Chain Address Pointer (CAP), which indicates the address of the next chain instruction.

The DSM requests instructions from memory using a 16 bit local address formed using information in the Control Memory. The adapter on the Master VPM then performs an address conversion, using page set 0, to obtain the absolute address. If the address is not located on the master VPM’s OBM, an MBUS operation can be used to transfer the requested data to the master VPM and back to the requesting DSM. The DSM, therefore, has the capability to reach any memory addressable by the VPM.

The DSM also has the same capability as standard I/O modules of executing instructions sent as part of the command word over the XBUS. These commands are sent when the VPM executes an I/O command instruction. They can be broadcast to all I/O modules or addressed directly to an individual module and are used primarily to set or clear I/O events. All of the XBUS commands that apply to the DSM are listed in Table 4.

OPCODE								a				m				FUNCTION CODE								
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	Broadcast
0	0	0	0	X	X	X	X	X	X	X	X	X	X	X	X	1	1	0	1	1	1	0	1	Set Boot Enable
0	0	0	0	X	X	X	X	X	X	X	X	X	X	X	X	1	1	0	1	1	1	0	1	CLR Boot Enable
0	0	0	1	X	X	X	X	X	X	X	X	X	X	X	X	1	1	0	1	1	1	0	1	Bit Restart
X	1	1	0	0	0	0	0	X	X	X	X	X	0	0	0	1	1	0	1	1	1	X	1	Master CLR
X	1	1	0	0	0	0	0	X	X	X	X	X	1	0	0	1	1	0	1	1	1	X	1	Set EIE
X	1	1	0	0	0	0	0	X	X	X	X	X	1	0	1	1	1	0	1	1	1	X	1	CLR EIE
X	1	1	0	0	0	0	0	X	X	X	X	X	1	1	0	1	1	0	1	1	1	X	1	Set Class III Enable
X	1	1	0	0	0	0	0	X	X	X	X	X	1	1	1	1	1	0	1	1	1	X	1	Clear Class III Enable
1	1	1	0	1	0	1	1	X	X	X	X	X	X	X	X	1	1	0	1	1	1	X	1	Set Map Event
																								Nonbroadcast
X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	0	1	0	0	P	P	P	P	Set XCMD Notice
X	1	1	0	0	0	0	0	X	X	X	X	X	0	0	0	1	X	0	0	P	P	P	P	Set CXMC Notice
1	1	1	0	1	1	1	1	X	X	X	X	1	0	0	0	1	X	0	0	P	P	P	P	CLR Map Event
X	1	1	0	0	0	0	0	X	X	X	X	X	1	0	0	1	X	0	0	P	P	P	P	Set EIE
X	1	1	0	0	0	0	0	X	X	X	X	X	1	0	1	1	X	0	0	P	P	P	P	Clear EIE
X	1	1	0	0	0	0	0	X	X	X	X	X	1	1	0	1	X	0	0	P	P	P	P	Set Class III Enable
X	1	1	0	0	0	0	0	X	X	X	X	X	1	1	1	1	X	0	0	P	P	P	P	Clear Class III Enable
1	1	1	0	1	1	1	1	X	X	X	X	0	1	0	0	1	X	0	0	P	P	P	P	CLR Class 2 DISC 0/4
1	1	1	0	1	1	1	1	X	X	X	X	0	1	0	1	1	X	0	0	P	P	P	P	CLR Class 2 DISC 1/5
1	1	1	0	1	1	1	1	X	X	X	X	0	1	1	0	1	X	0	0	P	P	P	P	CLR Class 2 DISC 2/6
1	1	1	0	1	1	1	1	X	X	X	X	0	1	1	1	1	X	0	0	P	P	P	P	CLR Class 2 DISC 3/7
1	1	1	0	1	1	1	1	X	X	X	X	1	0	0	0	1	X	0	0	P	P	P	P	CLR Class 4 DISC 0/4
1	1	1	0	1	1	1	1	X	X	X	X	1	0	0	1	1	X	0	0	P	P	P	P	CLR Class 4 DISC 1/5
1	1	1	0	1	1	1	1	X	X	X	X	1	0	1	0	1	X	0	0	P	P	P	P	CLR Class 4 DISC 2/6
1	1	1	0	1	1	1	1	X	X	X	X	1	0	1	1	1	X	0	0	P	P	P	P	CLR Class 4 DISC 3/7
1	1	1	0	1	1	1	1	X	X	X	X	1	1	0	0	1	X	0	0	P	P	P	P	CLR Class 7 DISC 0/4
1	1	1	0	1	1	1	1	X	X	X	X	1	1	0	1	1	X	0	0	P	P	P	P	CLR Class 7 DISC 1/5
1	1	1	0	1	1	1	1	X	X	X	X	1	1	1	0	1	X	0	0	P	P	P	P	CLR Class 7 DISC 2/6
1	1	1	0	1	1	1	1	X	X	X	X	1	1	1	1	1	X	0	0	P	P	P	P	CLR Class 7 DISC 3/7

Table 4. XBUS Commands – VPM to DSM

J. COMPUTER CONTROL UNIT

The Computer Control Unit (CCU) is a laboratory support unit that interfaces with the AYK-14 via a maintenance support channel. It provides the ability to load programs, display memory contents, set breakpoints and run software. The current version of the support unit is an emulator of the original that can run on a PC using DOS. The emulator (CCU/E) provides the same basic functional capabilities as the original CCU.

The CCU provides an extremely useful interface for troubleshooting hardware and software, or for gaining a better understanding of the AYK-14's internal operations. The software can be executed one instruction at a time (single-step) or run to a predefined location. The contents of memory, including registers, control memory, and OBM, can be

displayed using appropriate commands. The contents of memory can be changed via CCU commands as well in order to insert instructions to test hardware or debug software. Because the CCU is connected to the AYK-14 through the Maintenance Support Channel, all I/O channels are available for use in testing. The channels can be connected to external hardware or connected to each other for testing.

III. DESIGN IMPLEMENTATION

Once the design has been sufficiently recovered to provide a detailed understanding of the operation, the next step in the reengineering process is to begin the forward engineering of the new design. The difficulty in beginning the forward engineering process is deciding when the design has been adequately recovered. For a design as complex as the AYK-14, the design recovery could continue to reveal new aspects of the design almost indefinitely. However, once the design is thoroughly understood, the forward engineering process will actually provide more insight into the design than continuing with the design recovery. This is due to many factors including, first, that during the forward design process you continually become aware of what you do not know, which leads to more design recovery. And second, failures in the testing and validation of the new design will reveal and highlight misunderstanding of the recovered design.

This chapter will discuss the forward design process of the VPM adapter, specifically, the implementation process for the recovered design.

A. FORWARD ENGINEERING PROCESS

1. Field Programmable Gate Array

The first step in the forward engineering process is to determine how the new design is to be implemented. The target selected for this design was a Field Programmable Gate Array. This target was chosen due to the advantages of designing with FPGAs, specifically, the reduced time to develop and field products, the ability to maintain an open architecture, and the ability to design an entire system on a chip. (Ref. 1 p.29)

The ability to design a system on a chip is a key advantage to using an FPGA for this thesis. This is an advantage for two reasons. First, this thesis is the continuation of CDR Mike Croskrey’s thesis (Ref. 1) in which he designed the processor module of the VPM using an FPGA. The ability to design another module, the Adapter, and combine the two designs into a larger system that can be re-implemented is a key advantage. Second, because there will be additional designs that will need to be combined with this

design to finally reach the goal of reengineering the AYK-14, the FPGA provides the means to continue to expand the system.

Another important advantage to using an FPGA is the ability to rapidly prototype the new design. This is an advantage for reengineering because it provides the means to incorporate aspects of the design that were not recovered until the testing phase. This is essential in reengineering because there inevitably are aspects of the design that can not recovered from even the most detailed documentation.

2. VHSIC Hardware Design Language (VHDL)

In generating designs to be implemented onto FPGAs, there are multiple methods of describing the design dependant upon the software tools used for the design flow. These methods can be divided into graphical, code, or a combination of both. The graphical methods, such as schematic capture, provide a drag and drop approach which allows vendor specific components to be connected to form a design. The behavior of some of these components can be modified, and new components created, to allow addition design flexibility.

The advantage of the graphical method is the visual layout that it provides because it helps the user to visualize the ‘hardware’ being designed. Some of the disadvantages to this method are the limitations on components based on the contents of the vendor’s libraries, the inability to troubleshoot problems past the component or ‘black-box’ level, and lack of portability due to use of proprietary components. The lack of portability is the most important problem with the graphical methods because one of the goals of the reengineering process is an open architecture.

The code or programming method of describing a design has advantages and disadvantages as well. The advantages include the ability to design from the most primitive level and to modify the design at all levels of complexity. Another advantage is the portability of design due to the standardization of the design languages. The primary disadvantages of the programming approach are the difficulty visualizing the design due to the abstract nature of the code and the requirement to understand how the code is translated into a hardware implementation. An example of the difficulty of using software to describe hardware is the sequential operation of most software (i.e. C++) programs

versus the concurrent operation of hardware. For this thesis, the programming approach to hardware design was chosen for the advantages of portability, open architecture, and the ability to modify the design at all levels of complexity.

The VHSIC (Very High Speed Integrated Circuit) Hardware Design Language was used as the language to describe the design for implementation. VHDL is a hardware description language that was developed by the Department of Defense and given to the IEEE for standardization. It was designed to provide a language for describing hardware with a wide range of descriptive capability that would be independent of technology or design methodology.

3. FPGA Design Tools

The implementation of a design from a set of specifications through to hardware operation follows a specific set of steps, or design flow. When the target of the design is an FPGA, these steps are modified to include processes required to translate the design to a form than can be loaded onto the targeted chip. Figure 20 (Ref. 8, p33) illustrates the generic design flow in contrast to the FPGA specific design flow. The steps highlighted in grey in Figure 20 require the use of software tools to be performed. In addition to performing the necessary FPGA specific functions such as Map, Place, and Route, the tools provide additional editing and simulating functions that provide assistance in maintaining proper format and debugging code.

The reliance of the design process on software tools can cause difficulty and inefficiency in the FPGA Design process. The first cause of difficulty can originate from the functions that the software tools use to interpret the design and translate it into a form that can be simulated and implemented. These steps are complex and can generate errors that are often difficult to correct without a thorough understanding of the processes that are taking place. Another cause of difficulty can be the abstract level of designing with a hardware description language. Because the software tool creates the design from the language description, it can be difficult to visualize the 'hardware' implementation of the design. This, again, can cause difficulty in correcting errors in the design performance based on simulation.

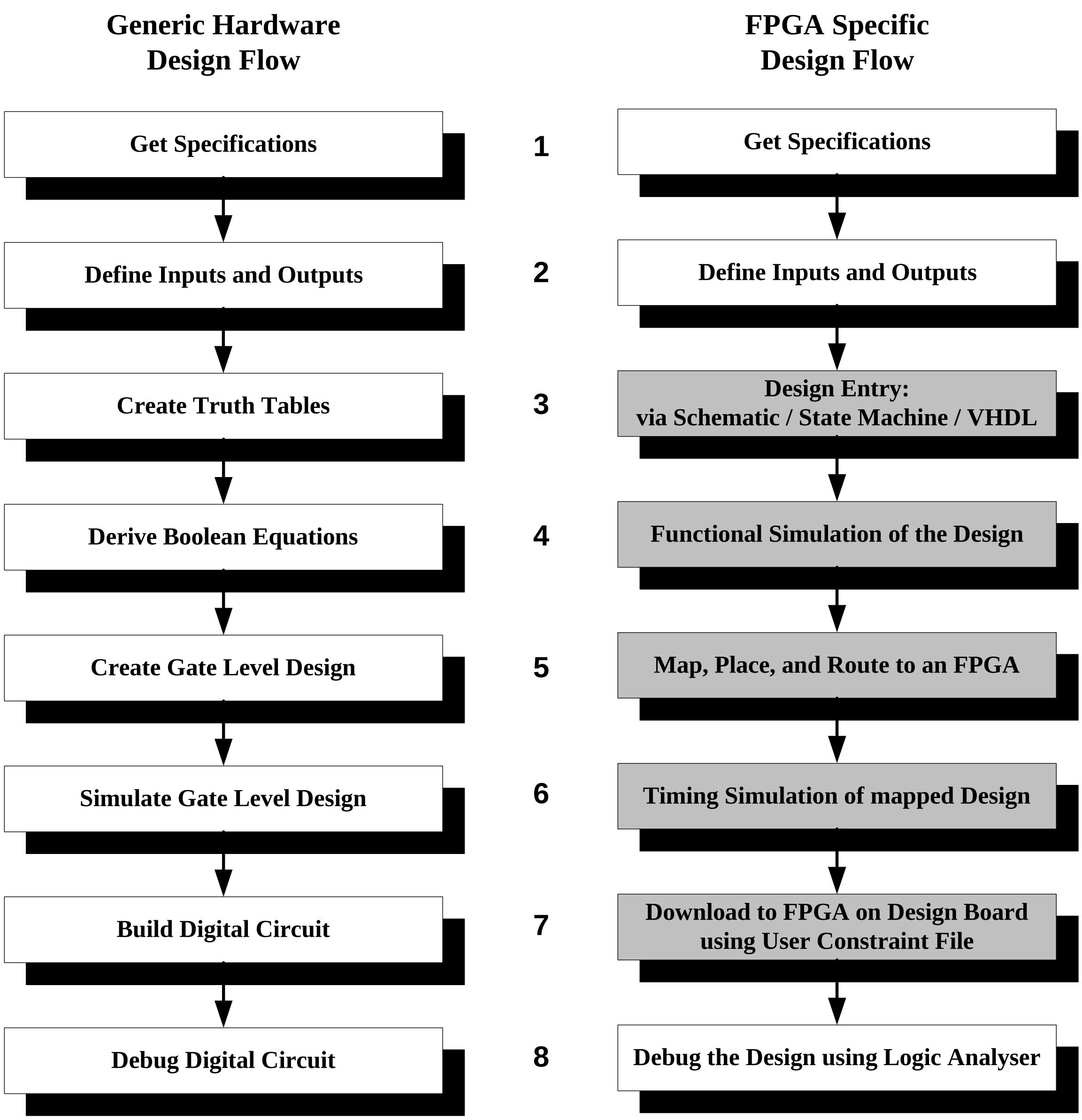


Figure 20. Hardware Design Flow

In the process of implementing the design for this thesis, four FPGA Design software tools were used. They included Xilinx Foundation, Xilinx ISE, ALDEC Active-HDL, and Synplicity Synplify Pro. Multiple tools were utilized during the design process to explore the advantages of each and to determine the most efficient method of getting from design to implementation. The majority of this thesis was created and implemented using Foundation primarily due to the author’s familiarity with the tool.

4. Finite State Machine Design

The design of complex hardware using powerful tools such as VHDL requires a methodology that allows extreme flexibility to meet varied requirements while providing an efficient and repeatable technique. The methodology that is generally regarded as the best way of meeting these goals is the Finite State Machine approach. In the finite state machine approach, the behavior of the design is divided into discrete states. In each state, the previous state and the input signals determine the next state. The values of all output signals are determined by the current state and the input signals. The state machine transitions from the current state to the next state based upon a synchronous signal or clock. There are many different approaches to designing state machines using VHDL. The method outlined in Reference 9 was used as the model for this thesis. Because of the complexity of the recovered design and the modular approach to reengineering it, the use of a very structured method to design the state machines was essential in order to create a design that was clear, readable, and easy to modify.

Finite State Machine

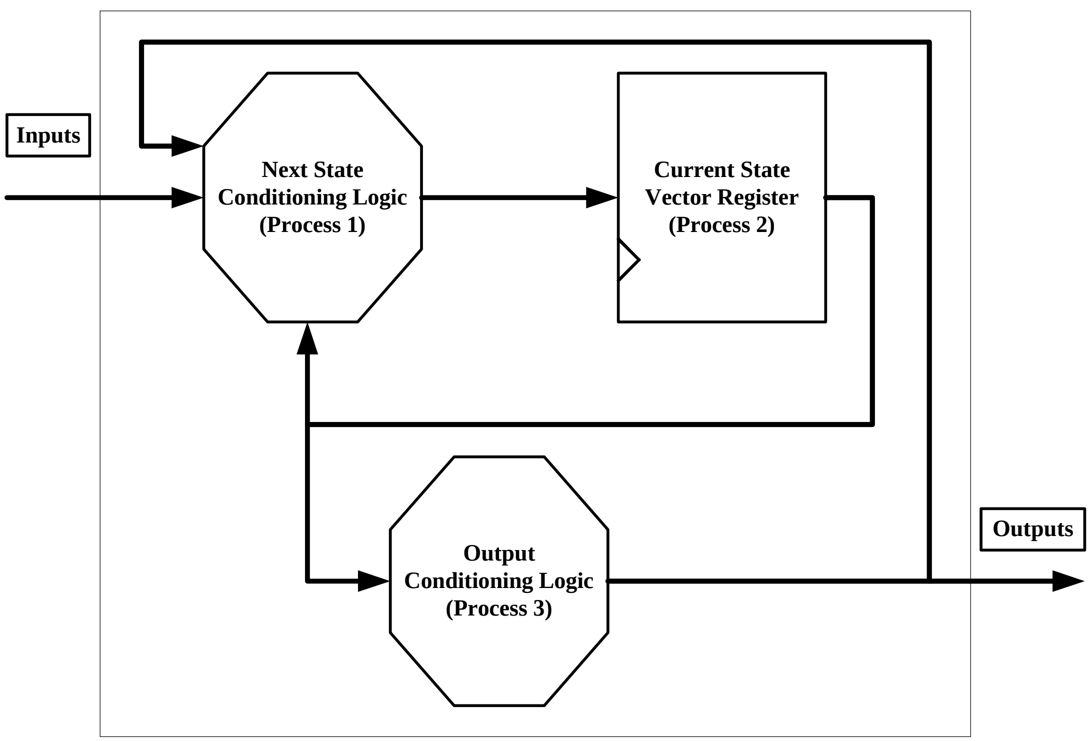


Figure 21. Finite State Machine Structure [After Ref. 9]

In this method, the state machine is divided into three blocks as shown in Figure 21. The Next State and Output Conditioning Logic blocks are combinatorial. This means that the outputs of these blocks change asynchronously based on the current state and changes in the inputs. The Current State register, in contrast, retains current state information and propagates next state information synchronously. This division of logic provides a simple structure to use in creating the state machine.

The first step in this method is to define the inputs and outputs. It is essential in this step to include all signals that can have any effect on or are affected by the component being designed. This is a step that is often repeated during the design process as the states and state transitions become more clearly defined.

The second step is to determine all of the possible states and state transitions. This is done through the creation of a state diagram. The state diagram is a tool used to illustrate the states and the state transitions in a logic format. It is an effective tool for visualizing the operation of a design, especially when using an abstract method of design description such as VHDL. Some of the software tools even have state machine editors that help create code from a diagram and vice-versa. Examples of state diagrams for each of the components designed are listed in Figures 26-30.

The third step is creating the three blocks of the state machine based upon the input, outputs, states, and state transitions. Each of these blocks is created as an individual process in VHDL. The first block is the Next State block (Process 1). This block is combinatorial and is dependant upon the current state, inputs, and outputs. The purpose of this process is to determine the next state that the state machine will transition into on the next clock cycle. The second block is the Current State register (Process 2). The purpose of this process is to advance the state machine to the next state, as determined by the Next State process, synchronously and also to handle system resets. The third block is the Output Conditioning block (Process 3). This block is also combinatorial and is dependant upon the current state and the inputs. The purpose of this block is to determine the outputs of the state machine.

5. Modular Approach to Overall Design

A modular approach was taken in the reengineering of the adapter because of the advantages of the combination of the State Machine method of hardware design along with the capability of VHDL to combine smaller components into a larger design. This approach allowed the design to be broken down into smaller, simpler designs based upon functionality. It also allowed the reusability of components and code to help make the design more understandable and easier to modify, much like the advantages to an object oriented approach in software design.

This modular approach also takes advantage of the rapid prototyping benefit of using FPGAs. This is done by adding functionality to the design simply through the addition of new components. The new design can quickly be tested at both the simulation level and actual hardware implementation level. The advantage here is that the design does not need to be completely defined early in the design process and that testing can continually be done to provide feedback and changes to the design. This is critical in the reengineering process since the goal is a design that has the same functionality as the replaced design, and therefore must be tested versus the original design's performance. As an example, in the adapter design, the memory interface was designed and tested as the first component. As additional components were created, they were tested individually and then in combination with the memory interface. This method allowed efficient and reliable detection of design errors.

B. TARGET FOR DESIGN IMPLMENTATION

The goal of the FPGA design process is to implement the design and load it onto a development board for testing and design validation. The three primary factors that were used in choosing a platform to economically implement this design were FPGA size, number of input / output ports, and memory capabilities. The development board chosen for this thesis was the Xilinx Virtex-E FPGA Development Kit from AVNET Design Services. The functional layout of the development kit is illustrated in Figure 22 (Ref. 10).

The FPGA used on this development kit is the Xilinx Virtex-E XCV1000E-6FG1156. The first reason this FPGA was selected is due to the author's experience and

familiarity with Xilinx FPGAs and Xilinx design software products. As previously mentioned, a thorough understanding with the software tools is critical to efficient FPGA design. A second reason for selection of this FPGA was the size of the chip in terms of number of logic gates as well as the number of off chip ports and chip speed. The Virtex-E XCV1000 has over 1,000,000 logic gates and 512 assignable off-chip ports, and is capable of operating at speeds as high as 200 MHz. The number of logic gates and the maximum operating frequency meet or exceed the capabilities of the targeted FPGA used in CDR Croskrey's design (Ref. 1, p34-36.) The XCV 1000 is therefore considered to have the additional capability available to expand the design to include the adapter control and interface. The number of available off chip ports (512) far exceed the number of required adapter Input / Output lines (152) which provides additional ports for the output of critical internal data and control signals for testing and troubleshooting.

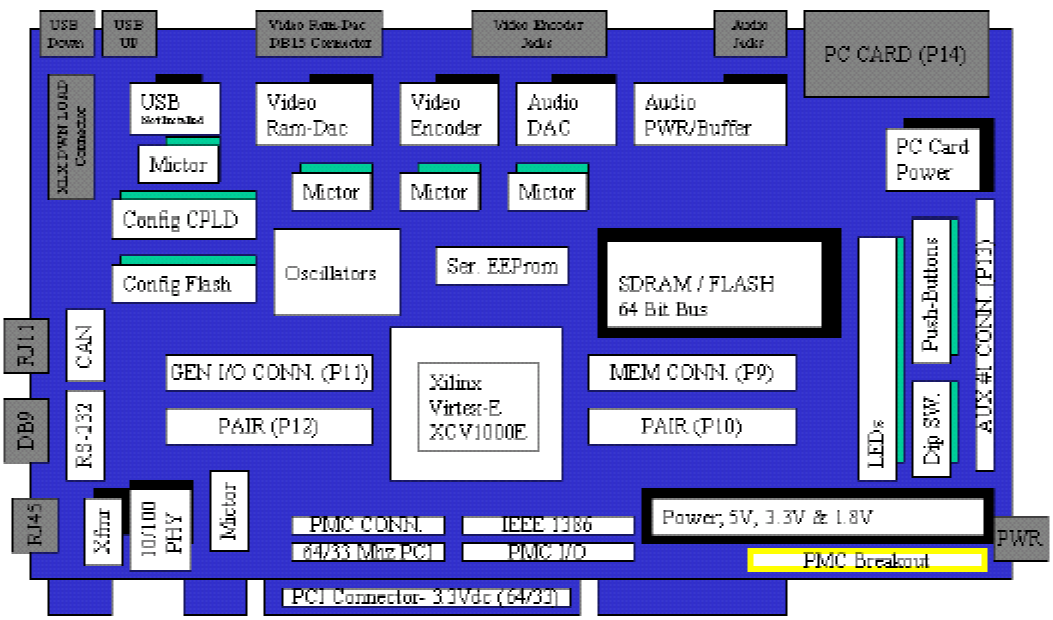


Figure 22. VIRTEX-E Development Board Functional Layout

The Virtex-E development board is configured with a 64-bit wide data bus for use of both on-board Flash and SDRAM memory. Common data and address buses are used to connect the FPGA with both Flash and SDRAM as well as I/O memory connectors. The SDRAM has a capacity of 64 Mbytes and the Flash has a capacity of 32 Mbytes. This memory configuration has both advantages and disadvantages for the implementation of this design. The first advantage is that the size of the memory is

sufficient to cover the entire OBM of the targeted design in either Flash or SDRAM. The second advantage is that the memory I/O connectors provide the means to either expand the memory capability or monitor memory activity. A disadvantage to the memory configuration is the use of SDRAM with no associated SDRAM controller. In order to use the SDRAM, a controller had to be designed and implemented to interface with the overall project design. The SDRAM also has latencies associated with reads and writes for non-sequential memory accesses. These disadvantages can be overcome with the addition of a cache memory component to make more efficient use of the existing memory configuration. However, the cache design is left to future students continuing on the implementation of the AYK-14.

C. COMPONENT DESIGN DESCRIPTION

The functions of the adapter were broken down into components, based upon function, in order to simplify the state machine design process and to enable reuse of code rather than duplication of effort. The components that make up the design are illustrated in Figure 23. The VHDL code for each component discussed is listed in Appendix E.

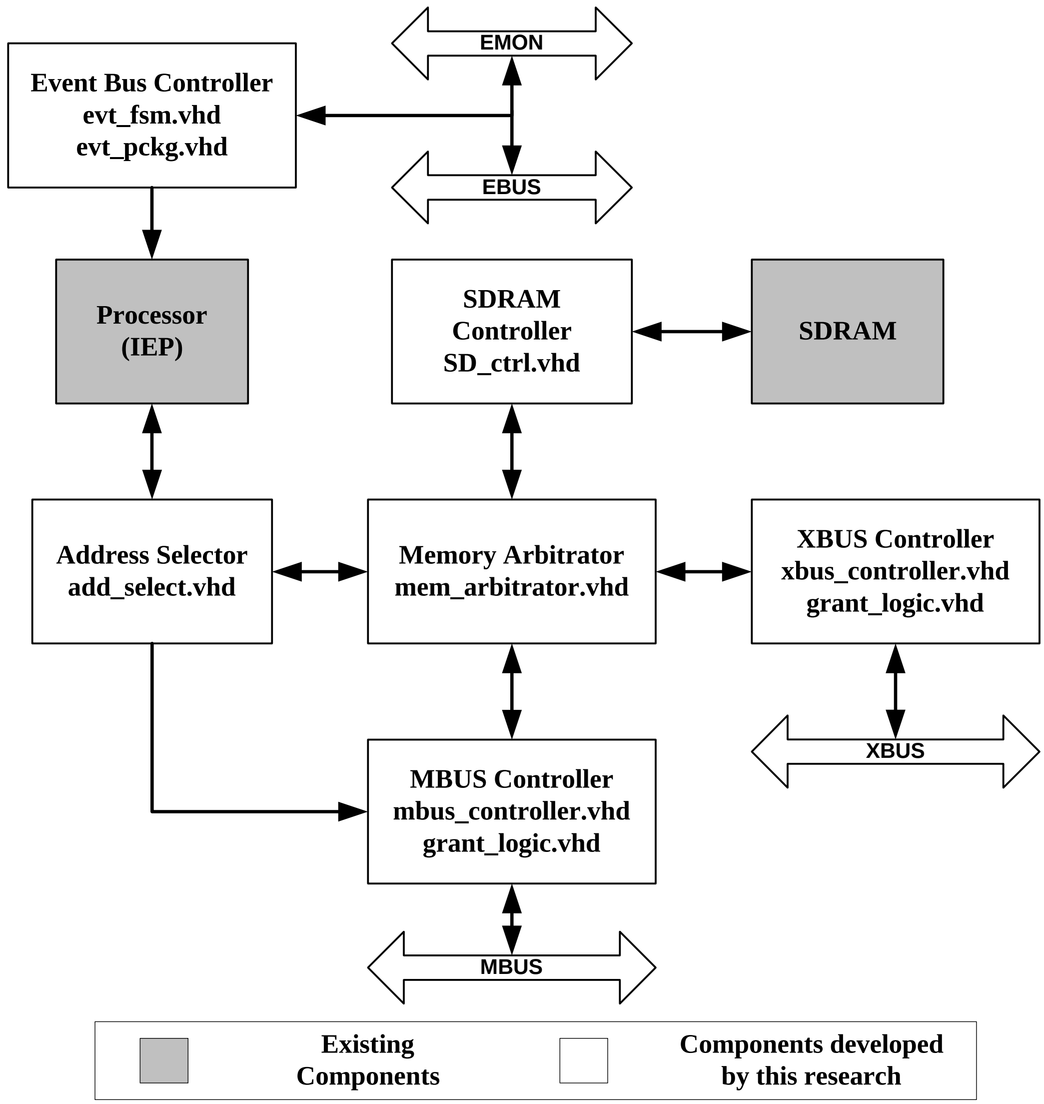


Figure 23. Adapter Design Components

1. SDRAM Controller

The memory available on the Virtex-E development board consists of Flash and SDRAM as outlined previously. The SDRAM was targeted to be used as the on board memory for the adapter design. A brief summary of the operation of this type of memory is presented in order to clarify the requirements of an SDRAM controller.

Synchronous Dynamic Random Access Memory (SDRAM) is a form of memory that is termed dynamic because it requires recharging or refreshing of its memory contents periodically, and termed synchronous because all signals are registered on the positive edge of the input clock signal. The components that make up the memory units

in an SDRAM consist of capacitors and transistors and the capacitors require recharging because they lose their charge when they are accessed or due to leakage over time. The development board uses four Micron 256 Mb Chips MT48LC16M16A2, each of which is internally configured as four 67,108,864-bit banks organized as 8,192 rows by 512 columns by 16 bits (Ref. 11). Based on this configuration each memory location, consisting of 16 bits, is defined by a bank, row, and column. Read and write accesses are burst oriented which allows sequential memory locations to be accessed in lengths of 1, 2, 4, or 8 locations. The internal SDRAM control logic maintains a loadable mode register that sets certain mode operation constraints. A functional block diagram of a single 256Mb SDRAM is illustrated in Figure 24. It should be noted that there are only 13 address lines because they are used for addressing either the column or the row depending upon the control signals present.

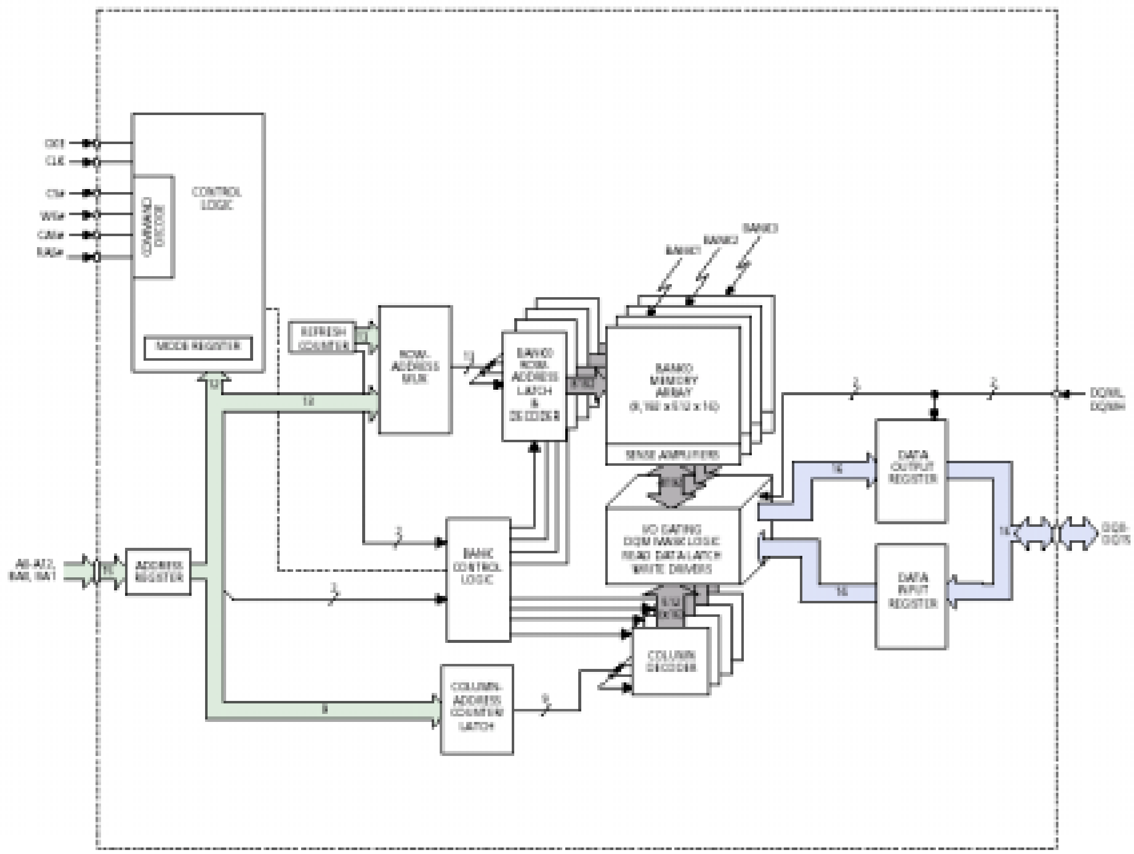


Figure 24. SDRAM Functional Block Diagram

The complexity of SDRAM operation requires a memory controller to be used in order to meet all the maintenance requirements of the chip including precharging before a memory access, periodic refreshing of all memory locations, and providing the control signals to read or write to memory. This allows memory accesses to be treated as independent of the memory source when creating the other components that require access to memory. This method also provides the capability to expand the design in the future to include a cache to increase system performance.

Due to the complexity required in the design of an SDRAM controller and the time constraints of the design process, the design for the SDRAM controller was adapted from existing designs. The design that was ultimately selected was the XSA SDRAM controller from the XESS Corporation. The original design for this controller was written in VHDL and targeted to a different development board. It was modified and tested to operate with the SDRAM configuration on the Virtex-E development board. The controller interface is illustrated in Figure 25.

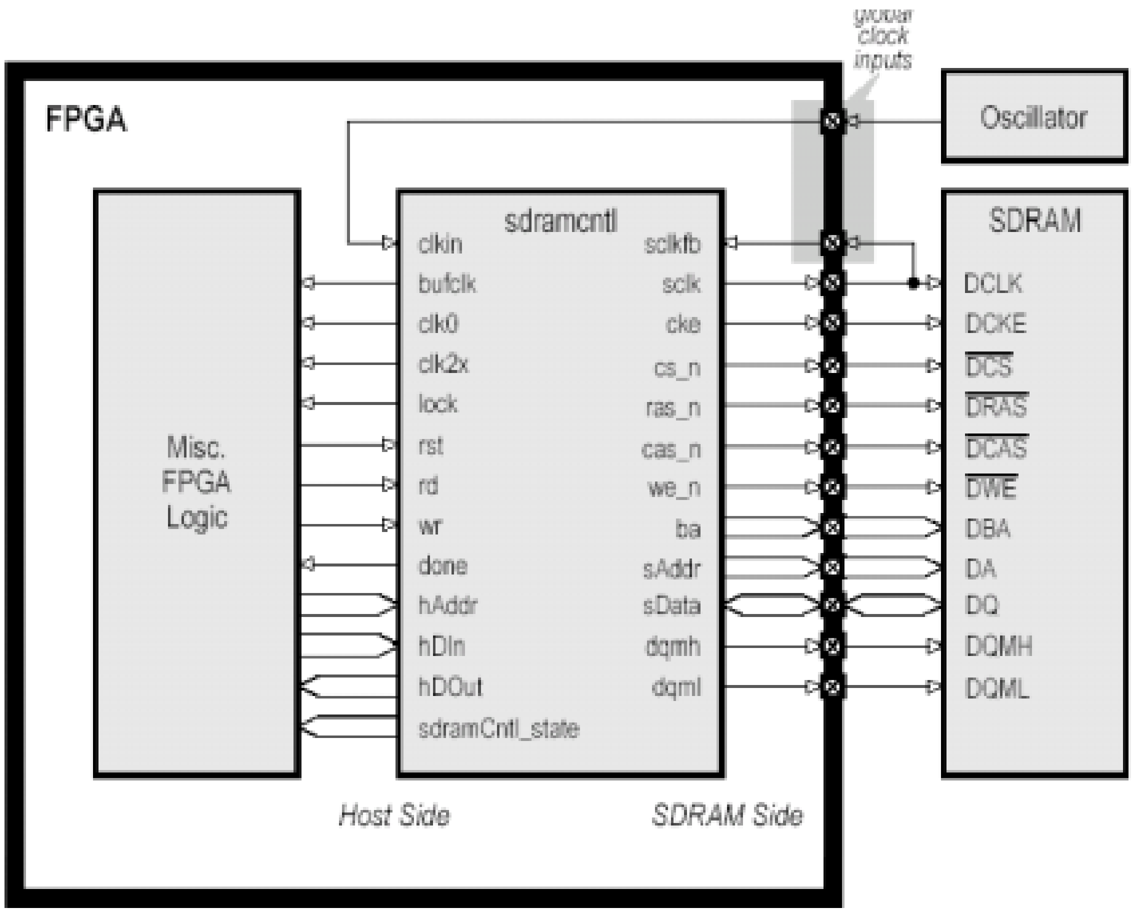


Figure 25. SDRAM Controller Interface

2. Memory Arbitrator

The Memory Arbitrator is the component that provides the interface between each memory user and the SDRAM controller. The three possible users of memory are the Processor, the XBUS, and the Memory Bus. The Arbitrator monitors requests for memory from the three users and grants use based on a rotating priority scheme. The scheme is based on the rotating scheme as described in MBUS arbitration. The rotation scheme insures that each component is allowed memory use at least one out of every three memory accesses. The priority is based upon the current user, the last user, and the users requesting access. The default priority is the Processor, the XBUS, and then the MBUS based upon the expected frequency of use.

The design is based on the three-process State Machine method previously discussed. The priority in each state is accomplished using if-then statements. Because

these statements are executed sequentially, levels of priority can be assigned through the order of the statements. Using this method, each state had a different order of priority of the two remaining states. The state diagram is shown in Figure 26. The following description of State Diagram symbology applies to all state diagrams shown in this thesis.

The names used in the state diagram are intended to reflect the names of the states and signals used in the VHDL code. The words attached to each arrow indicate the signals that are required to be true in order for the state transition to occur. If a signal is asserted low, the signal name will have a '_L' appended to it. If the condition to be met for transition is that a signal is NOT asserted, the signal name will be enclosed in parentheses. The boxes next to certain states contain the signals that are driven while in that state. The ampersand symbol (&) is used to indicate a logical AND of conditions to be met for signal transition.

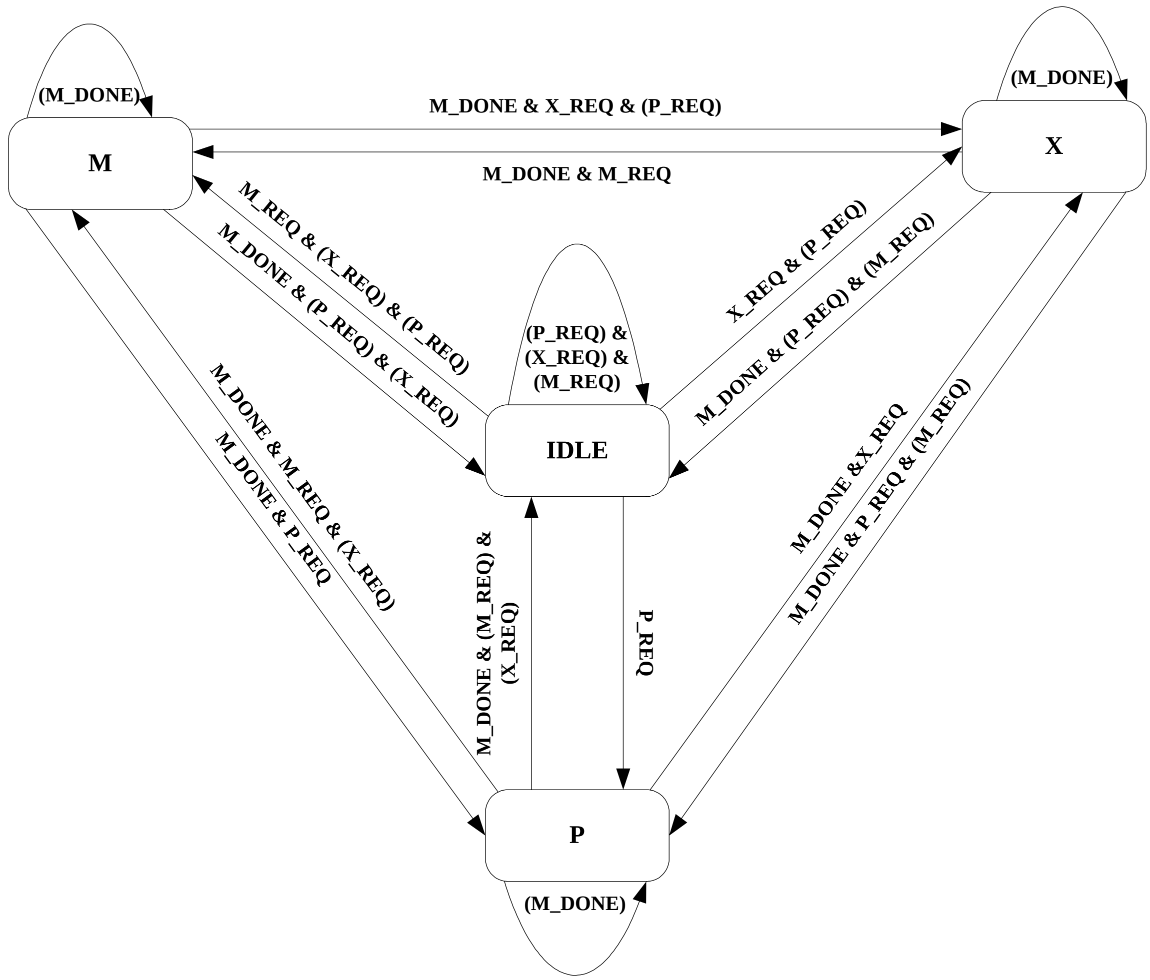


Figure 26. Memory Arbitrator State Diagram

3. MBUS Controller

The MBUS controller is the component that controls the Memory Bus interface between the processor, external users, and on board memory. Its primary function is to operate the bus control signals required by the MBUS protocol. This protocol includes all required control signal, timing requirements, parity generation, and error detection. It requests use of the on board memory for reads or writes by external bus users. It also operates as the MBUS arbitrator by determining the priority user and granting usage of the bus.

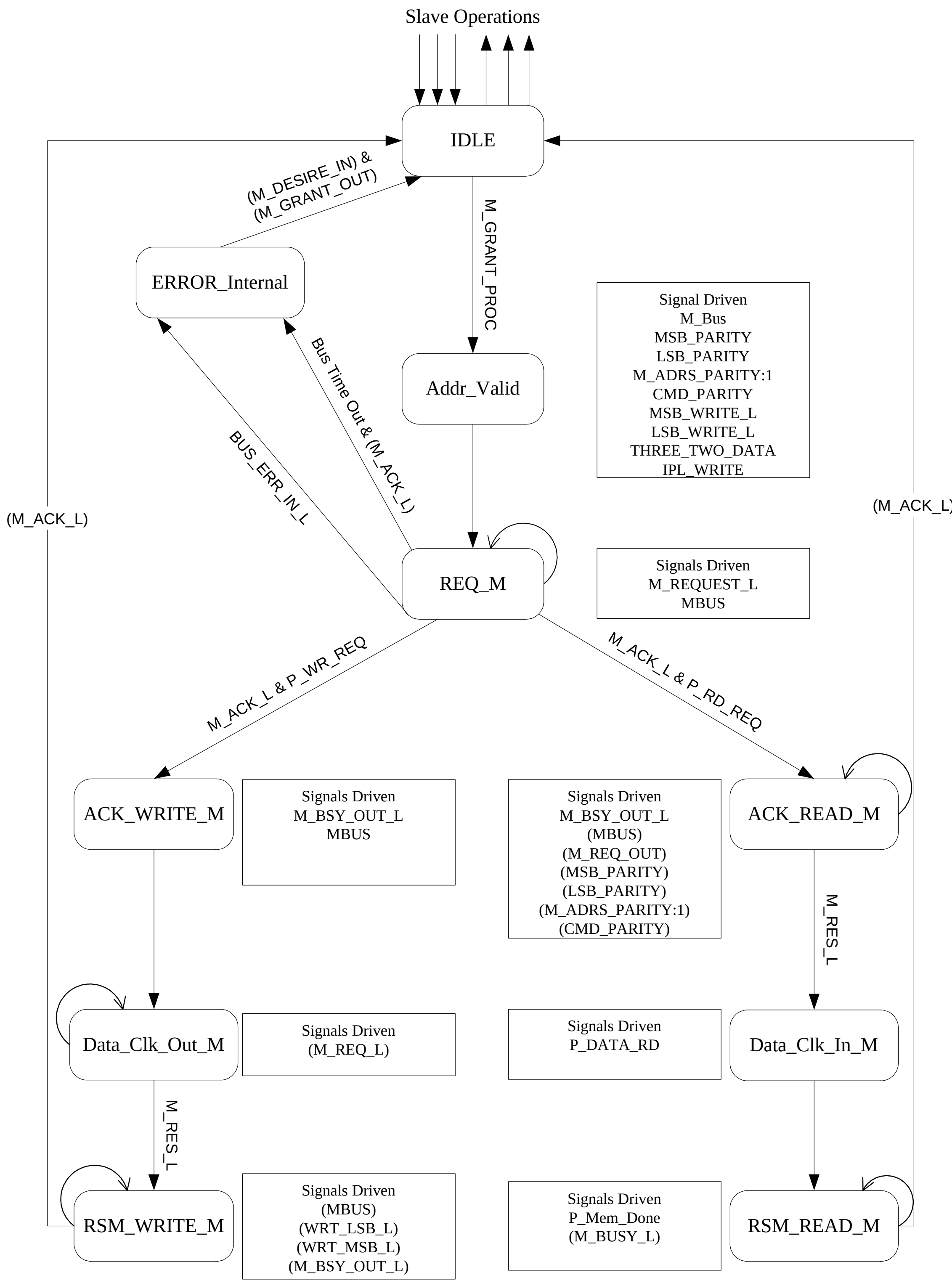


Figure 27. MBUS Controller State Diagram (Master)

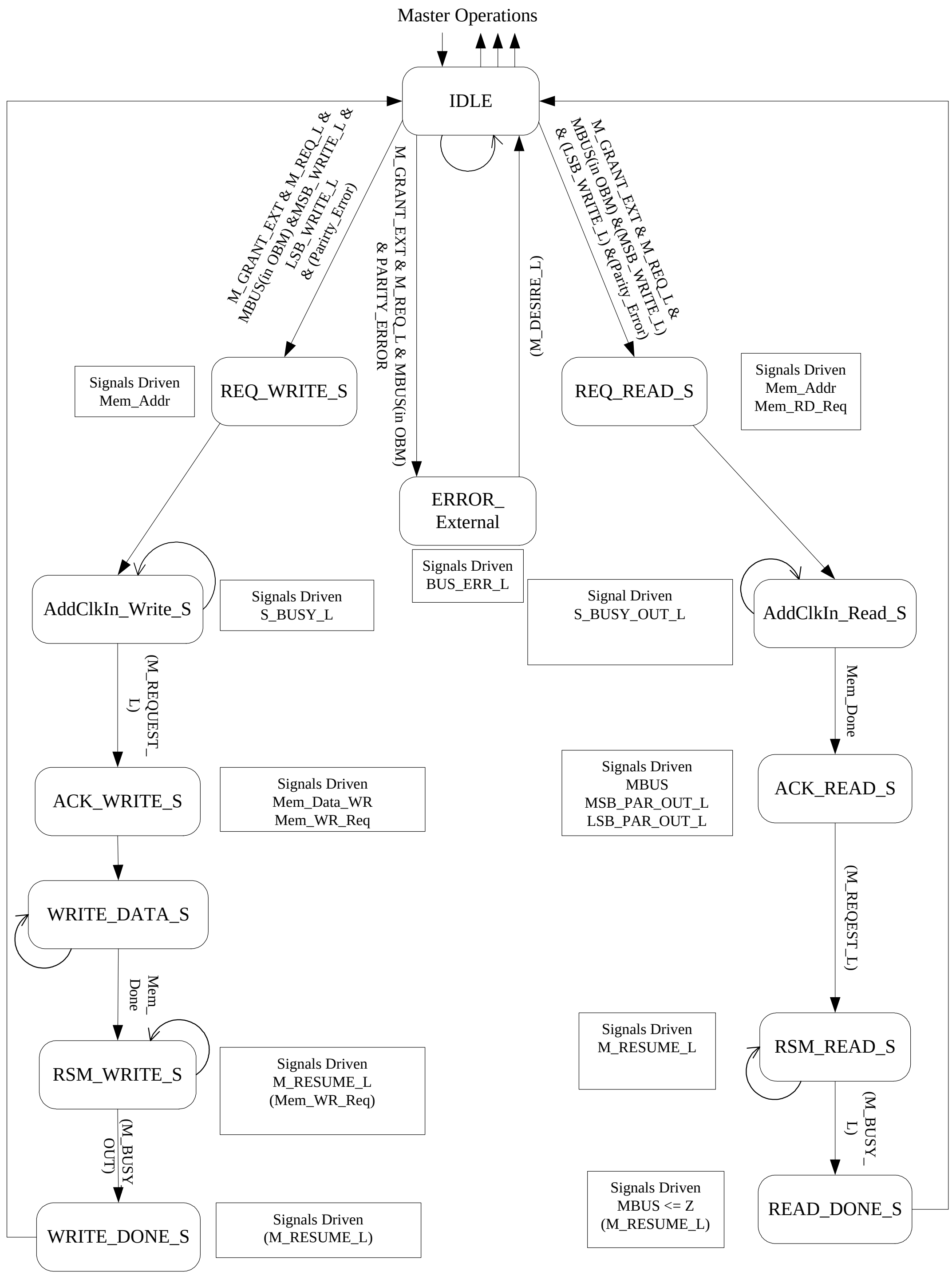


Figure 28. MBUS Controller State Diagram (Slave)

The design is based on the three process state machine previously discussed. There are four basic types of operation that can occur on the MBUS. The first two are either a read or a write by the processor and the second two are either a read or write by

an external user. The state diagram for the controller is illustrated in Figures 27 and 28. The diagram was split into two figures for clarity of state flow with the Idle state serving as a common State between the Figures. Figure 27 illustrates the states for bus usage by the processor and Figure 28 illustrates bus usage by an external user.

The MBUS Controller design has two components included to provide MBUS usage arbitration and parity generation. The component Grant Logic performs the bus arbitration in a rotating priority scheme. It is a three process state machine with priority logic similar to the Memory Arbitrator. The function of arbitration was accomplished using a component in order to facilitate design reuse. The component OddParityGen is an odd parity generator used to generate parity for transmission or for comparison with received parity to provided error detection.

4. XBUS Controller

The XBUS controller is the component that controls the XBUS interface between the processor, external users, and on board memory. Its primary function is to operate the bus control signals required by the XBUS protocol. It requests the use of on board memory for reads and writes by external users. It also operates as the XBUS arbitrator by determining the priority user and granting usage of the bus.

The design is based on the three process state machine previously discussed. The XBUS operation was divided into three basic types of operation, output, input, and broadcast. The users were also divided into two groups, the processor and the external users. The state diagram for the controller is illustrated in Figures 29 and 30. The diagram was split into two figures for clarity of state flow with the Idle state serving as a common State between the Figures. Figure 29 illustrates the states for bus usage by the processor and Figure 30 illustrates bus usage by an external user.

The XBUS controller has a component included to provide bus user arbitration. The component X_GRANT_LOGIC performs the arbitration in a rotating priority scheme similar to the Memory Arbitrator component. This component ensures each users has control of the bus at least once every seven uses (there are seven users of the XBUS).

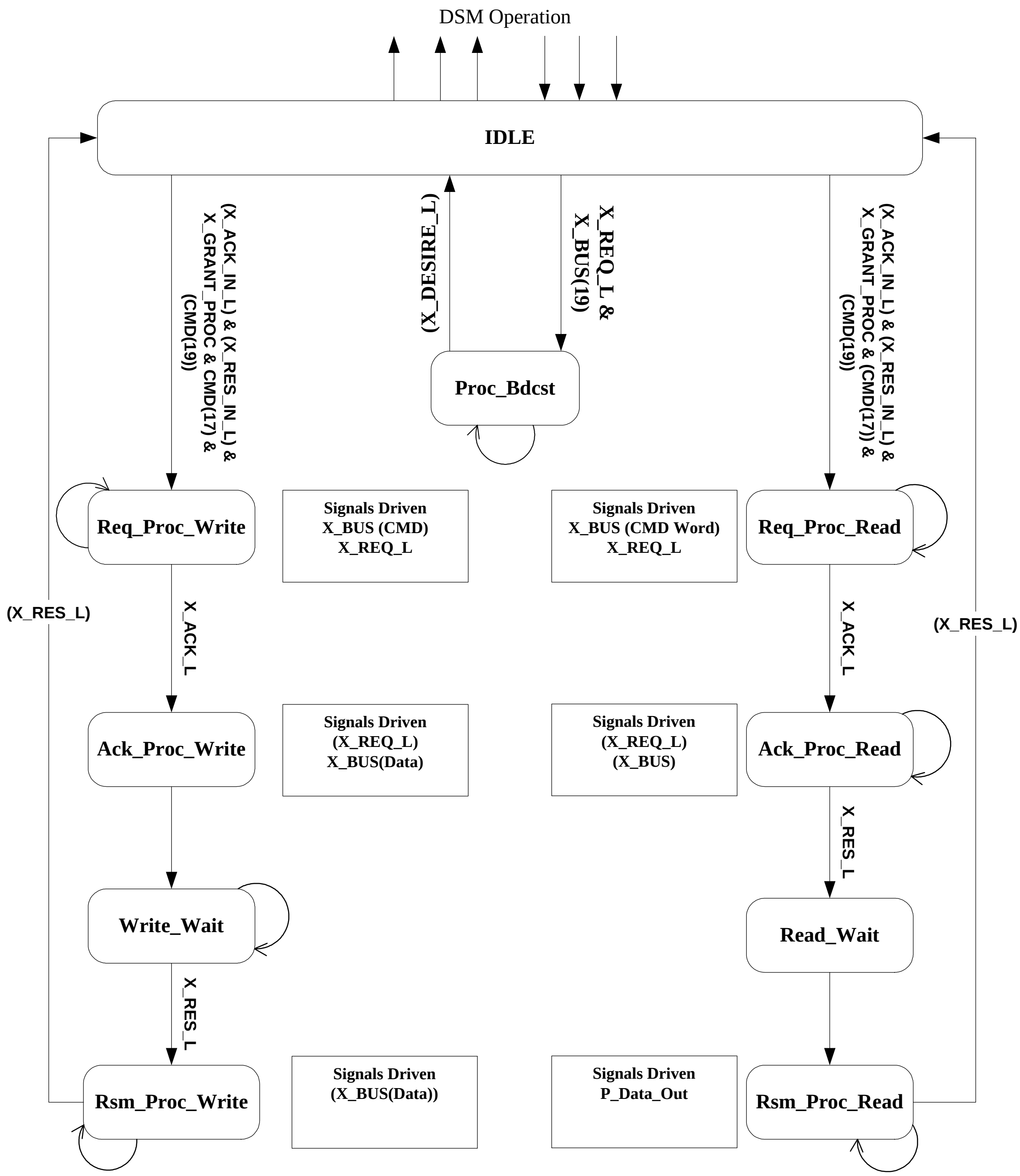


Figure 29. XBUS Controller State Diagram (Processor)

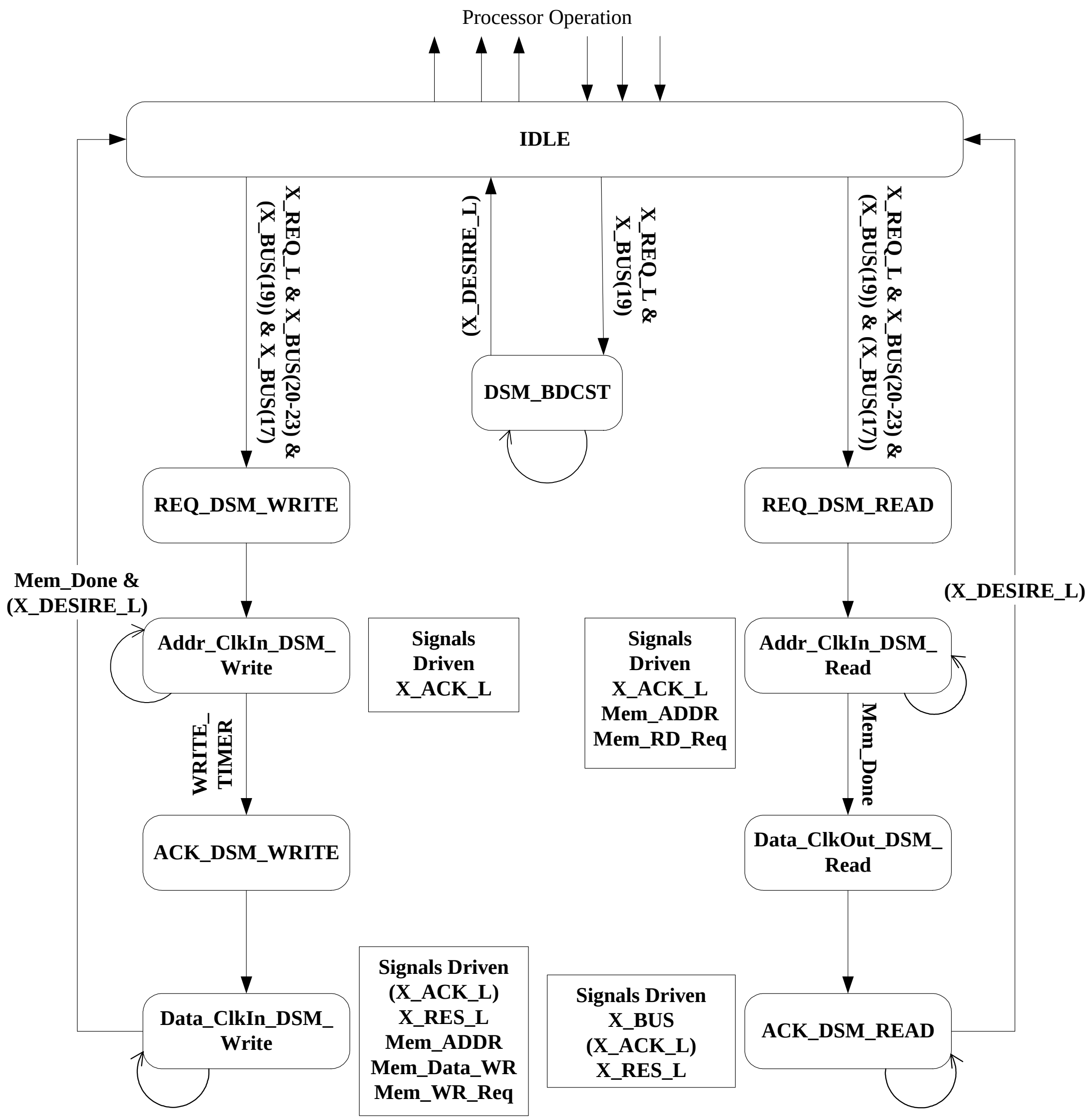


Figure 30. XBUS Controller State Diagram (DSM)

5. Event Bus Controller

The Event Bus Controller is the component that determines the highest active event using the event polling sequence on the Event and Event Monitor busses. It also generates the Event vector to notify the processor of the highest active event. The design was based on the three process state machine. It requires a timer in order to meet the Event bus protocol. The timer logic is based upon the operating frequency of the intended design. If the design is targeted to a faster clock frequency, only one constant needs to be

updated in the design to allow the component to continue to meet the timing constraints. The state diagram for the Event Bus Controller is illustrated in Figure 31.

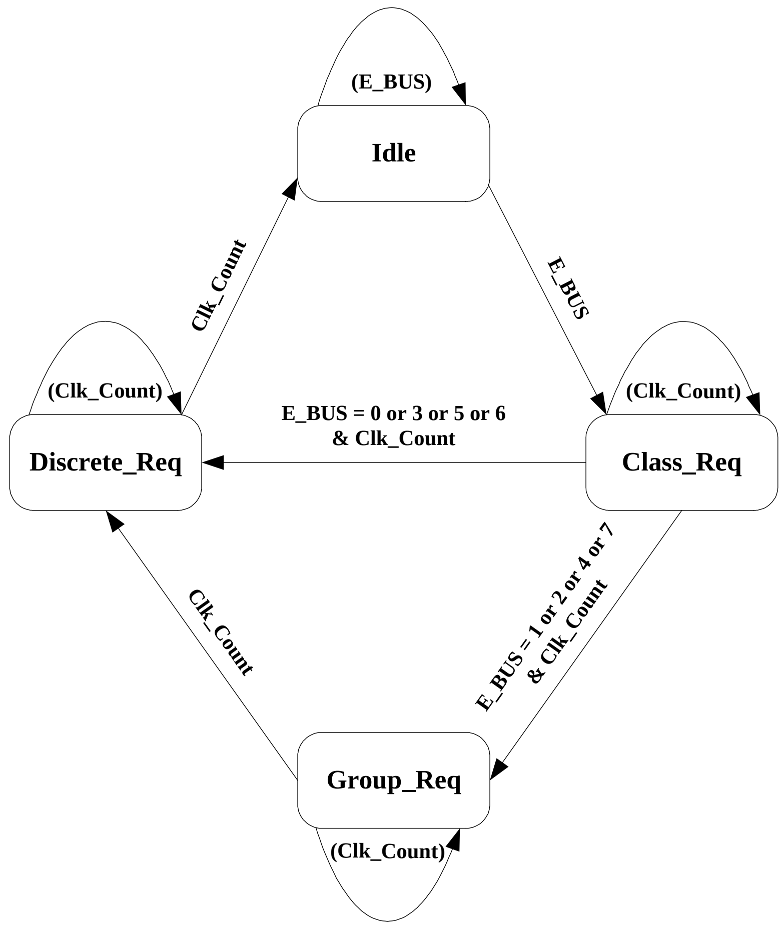


Figure 31. Event Controller State Diagram

6. Top Level Design Interface

The Top Level Design Interface is simply the component that combines all of the previous components into a single entity. It connects all of the components, including the Processor, via internal signals as shown in Figure 23. It also connects all of the appropriate signals to input or output ports.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. CONCLUSIONS

There were three primary goals that this thesis set out to achieve. The first was the reengineering of the adapter module on the VPM of the AYK-14. This goal was a milestone toward the larger goal of validating the theory that a binary compatable processor, designed using FPGA technology, would be a viable solution to deal with the growing legacy avionics problem in the Department of Defense. In terms of this second goal, this thesis attempted to continue the process of reengineering the processor begun with CDR Croskrey's work in his Master's thesis. The third goal of this thesis was to create a reference that summarized the operation of the AYK-14 emphasizing VPM to I/O module communication.

In terms of this first goal, this thesis succeeded in the reengineering process to the level of simulating a design whose performance matched the operation of the VPM adapter based upon design documentation. It should be stressed that this performance comparison is based on performance descriptions and diagrams from the design documentation. This is stressed because an important lesson learned was the need for actual hardware for use in testing early in the design recovery process.

The reason for this requirement for hardware is the difficulty in recovering a design from documentation alone. There was a large amount of documentation available for the AYK-14. However, due to the AYK-14's complexity, age, and numerous upgrades over its lifecycle, the documentation did not cover every aspect of the design to the level required for a complete design recovery. An AYK-14 was available with a CCU testing unit during the early stages of the design recovery but due to the complexity of the CCU interface, it did not provide useful information until the design was more clearly understood. The testing that was needed to aid in the design and validate the simulated design was a sampling of all bus operation using a logic analyzer.

In terms of the second goal, this thesis demonstrated that a complex design could be recovered and reengineered using the tools available to design FPGAs.

The third goal of this thesis was accomplished as a byproduct of the design recovery process. The difficulty in creating a summary of the AYK-14 operation without full and detailed testing is that the summary is only as valid as the documentation it was taken from. However, because of the numerous and varied sources of information, this document will at least serve as a starting point for a more detailed study. It will also clarify concepts regarding the I/O system that are difficult to understand without a detailed understanding of AYK-14 operation.

APPENDIX A: DOCUMENTATION LIST FOR THE AYK-14

Reference	Document Title	Prepared By	Date
1	VHSIC Processor Module Equipment Specification, Standard Airborne Computer AN/AYK-14 (V)	Control Data Corporation	February 1990 with SCN 1
2	Design Guide for AN/AYK-14 (V) Input/Output Modules	Control Data Corporation	July 1985
3	VHSIC Processor Module (VPM) Interface Design Specification (IDS)	Control Data Corporation	March 1993
4	VHSIC Processor Module (VPM) to Discrete and Serial Module (DSM) Interface Design Specification (IDS)	Control Data Corporation	October 1986
5	Discrete and Serial Module (DSM) Equipment Specification	Control Data Corporation	August 1988
6	Cooling Design Data for AN/AYK-14 (V) Configurations	Control Data Corporation	May 1984
7	AN/AYK-14 (V) Preplanned Product Improvement Story (Presentation Slides)	Control Data Corporation	Est. 1985
8	AN/AYK-14 (V) Programmers Reference Manual, Volume 1, General Reference Information	Computing Devices International	April 1995
9	AN/AYK-14 (V) Programmers Reference Manual, Volume 2, Input/Output Channel Information	Computing Devices International	April 1995
10	AN/AYK-14 (V) Programmers Reference Manual, Volumes 3, Instruction Execution Timing Information	Computing Devices International	April 1995
11	AN/AYK-14 (V) Standard Airborne Computer Set Test Requirements Document for VHSIC Processor Module with Appendix A, B, C, D, E	Computing Devices International	November 1994
12	AN/AYK-14 (V) Navy Standard Airborne Computer Technical Description	Computing Devices International	Unknown
13	VHSIC Program Equipment Specification Type 8 Chassis	Computing Devices International	April 1992
14	VPM Microcode Language Reference Manual	Computing Devices International	July 1992
15	Schematic Diagram VPM25B B-Side	Computing Devices International	Initial Release: October 1995
16	Software Design Specification for the Extended Memory Reach (EMR) Firmware Update on the VHSIC Processor Module (VPM)	General Dynamics Information Systems	April 1998
17	Interface Design Specification CPU/IOP/EIOP/SCP/VPM to Computer Control Unit (CCU)	General Dynamics Information Systems	March 1980
18	Schematic Diagram VHSIC Processor Module VPM25	General Dynamics Information Systems	Initial Release: August 1992
19	Programmers Reference Card	McDonnell Douglas	January 1990
20	Absolute Source-Object Listing for N1FG0410 F/A-18 Mission Computer Operational Flight Program	NAWC-AD China Lake, CA	September 1999
21	F/A-18C Mission Computer AYK-14 Instruction Usage	NAWC-AD China Lake, CA	June 1999
22	AN/AYK-14 (V) CCU Emulator VAX and PC User Manual	NAWC-AD Patuxent River, MD	August 1996
23	Discrete Serial Module (DSM) Training Course	Naval Aviation Depot, Norfolk, VA	Unknown

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B: DIRECT AND POLLED EVENTS

Event Class Discrete	**Event Description**
Event Class 0	
Discrete 0	Internal power down/power fail
Discrete 1	External power down
Discrete 2	Internal PCM thermal/thermal fault
Discrete 3	External PCM thermal
Discrete 4	MBUS timeout
Discrete 5	XBUS timeout
Discrete 6	Embedded power fail
Discrete 7	-
Event Class 1	
Discrete 0	Even channel RTCMD
Discrete 1	Even channel ODR
Discrete 2	Even channel IDR
Discrete 3	-
Discrete 4	Odd channel RTCMD
Discrete 5	Odd channel ODR
Discrete 6	Odd channel IDR
Discrete 7	-
Event Class 2	
Discrete 0	Even channel UCR/restart
Discrete 1	Even channel EIR
Discrete 2	Even channel ODR
Discrete 3	Even channel IDR
Discrete 4	Odd channel UCR
Discrete 5	Odd channel EIR
Discrete 6	Odd channel ODR
Discrete 7	Odd channel IDR
Event Class 3	
Discrete 0	Microevent 1/stop
Discrete 1	Watchdog timer
Discrete 2	File multiple bit error
Discrete 3	PA event/SIOP & ERI
Discrete 4	CCU event
Discrete 5	PB event/SYNC & IOCR & EII
Discrete 6	External interrupt event 2
Discrete 7	External Stop/step/run

Event Class Discrete	**Event Description**
Event Class 4	
Discrete 0	Even channel MAP
Discrete 1	Even channel OCR
Discrete 2	Even channel ICR
Discrete 3	Even channel EIR
Discrete 4	Odd channel MAP
Discrete 5	Odd channel OCR
Discrete 6	Odd channel ICR
Discrete 7	Odd channel EIR
Event Class 5	
Discrete 0	Recoverable error
Discrete 1	Operand memory error
Discrete 2	Instruction memory error or MCM parity fault
Discrete 3	Hardware fault warning
Discrete 4	External interrupt event 3
Discrete 5	Microevent 0
Discrete 6	Hardware fault (BIT error)
Discrete 7	Module overtemp event
Event Class 6	
Discrete 0	Memory protect fault
Discrete 1	RTC lower overflow
Discrete 2	Monitor clock overflow
Discrete 3	-
Discrete 4	System reset
Discrete 5	Initial program load
Discrete 6	External event 0/IPI 0
Discrete 7	External event 1API 1
Event Class 7	
Discrete 0	Even channel ERI or microevent 2
Discrete 1	Even channel Ell
Discrete 2	Even channel OCI
Discrete 3	Even channel ICI
Discrete 4	Odd channel ERI
Discrete 5	Odd channel EII
Discrete 6	Odd channel OCI
Discrete 7	Odd channel ICI

APPENDIX C: I/O INSTRUCTIONS

COMMAND CHAIN INSTRUCTIONS

Mnemonic	Hex	Instruction
ACR	EO 0 0	CHANNEL CONTROL Master clear all channels
ACR4 CCR0,4	E004	CHANNEL CONTROL Enable external 4 interrupts, all channels
ACR5 CCR0,5	EO 0 5	CHANNEL CONTROL Disable external interrupts, all channels
ACR6 CCR0,6	EO 0 6	CHANNEL CONTROL Enable class III interrupts, priorities 2,3,4
ACR6 CCRa,6	EO a 6	CHANNEL CONTROL Enable class III interrupts, priorities 2,3,4 for channels with priority less than channel a
ACR7 CCR0,7	EO 0 7	CHANNEL CONTROL Disable class III interrupts, priorities 2,3,4
ACR7 CCRa,7	EO a 7	CHANNEL CONTROL Disable class III interrupts, priorities 2,3,4 for channels with priority less than channel a
CCRa,12	EO a C	CHANNEL CONTROL Enable channel a external interrupts
CCRa,13	EO a D	CHANNEL CONTROL Disable channel a external interrupts
CCRa,14	EO a E	CHANNEL CONTROL Enable channel a,class III, priorities 2,3,4
CCRa,15	EO a F	CHANNEL CONTROL Disable channel a, class III; priorities 2,3,4
CCRa,8	EO a 8	CHANNEL CONTROL Master clear channel a

COMMAND INSTRUCTIONS

Mnemonic	Hex	Instruction
ICKa,y	E6 a 2	INITIATE INPUT CHAIN Y->Channel a Chain Pointer; initiate input chain
OCKa,y	E6 a 6	INITIATE OUTPUT CHAIN Y->Channel a Chain Pointer; initiate output chain
TOCKa,y,m	E6 a F	INITIATE OUTPUT CHAIN Y is chain table pointer; initiate tabular output chain
RIMa,y,m	EB a m	READ CONTROL MEMORY Channel a (CMm)->Y
SICRa,m	F8 a m	SET AND CLEAR DISCRETES Set or clear channel a discrete function
SIOPm,y	FC - m	START SLAVE m:0->EIOP/slave VPM/slave SCP SR1:12,Y->EIOP/slave VPM/ slave SCP P if m=0 or 1
SSTa,y,m	FB a m	STORE STATUS Channel a status bits per m->Y
WIMa,y,m	E7 a m	WRITE CONTROL MEMORY (Y)-> Channel a CMm
XIMa,y,m	FE a m	EXCHANGE CONTROL MEMORY Channel a (CMm)->Y;(Y+1)->Channel a CMm rf m=2 or 6. If m#2or6,1/O instruction fault.

COMMAND CHAIN INSTRUCTIONS

Mnemonic	Hex	Instruction
BJm,y	FD - m	BIT JUMP Y->CAP if(CM3):m=1
CSIRm	F8 0 m	SERIAL INTERFACE CONTROL Set or clear discrete function
CSSTy,m	FB - m	STORE STATUS Status bits per m->Y
HCR	EC 0 0	HALT CHAIN Halt chaining, a even
IMa,y,m	E2 a m	INITIATE MESSAGE Y->CMm; initiate message activity
IOa,y	E3 a 0	10 FUNCTION a (Y<Y+1)->BCW,BAP; initiate transfer
IPR	EC1 0	INTERRUPT PROCESSOR Generate chain interrupt, a odd
LCM m,y	E7 0 m	LOAD CONTROL MEMORY (Y)->CMm
LCMKm,y	E6 0 m	LOAD CONTROL MEMORY Y->CMm
SCMm,y	EB 0 m	STORE CONTROL MEMORY (CMm)->Y
SFy	EF 1 0	SET FLAG 1->y:15,14, a odd
SFSCm	F4 0 m	SEARCH FOR SYNC Perform function(s) assigned to m-bits
SJMCa,y	F2 a 0	SERIAL JUMP ON MET CONDITION Y->CAP
XCMm,y	FE - m	EXCHANGE CONTROL MEMORY (CMm)->Y; (Y+1)->CMm
ZFy	EF 0 0	ZERO FLAG 0->Y:15,14, a even

THIS PAGE INTENTIONALY LEFT BLANK

APPENDIX D: XBUS COMMAND WORDS

The first section of Appendix D describes the processor to I/O channel interaction on the XBUS (I/O Bus) for various I/O software instructions. Each software instruction is listed along with the associated XBUS activity directed to the I/O channel.

Notes:

1. For I/0 channels, bits 17-23 of the IOBUS are defined only during the CONTROL portion of the cycle. Bit 16 is as shown in the table during the CONTROL time, and is driven to a logic 0 by the I/0 module during the DATA portion of the IOBUS cycle

2. For I/0 channels, IOBUS bit 18 is always a logic 0. It must be decoded.

3. For I/0 channels, IOBUS bit 19 is a logic 0 for all non-"BROADCAST" operations.

4. On the IOBUS, complement polarity is used so that a logical 1 is represented by a ground potential.

5. The values shown in this table are logic true values.

6. An "x" in the value for XC or XO implies that those bits are indeterminate and the I/0 hardware shall not decode them.

7. The hardware bit-numbering scheme is used in this table (bit 0 = MSB).

8. The comment "Accept data word (XO)" simply implies that the I/0 channel must respond with X-Acknowledge and X-Resume signals. The data is not necessarily used.

9. The value (yyyy) refers to the contents of the memory location whose address is yyyy.

10. The "Priority Number" is:
    1. The priority number of the chain program being executed in the case of a chain command, or
    2. The priority number of the channel whose logical number is "a" in the case of a command cell.

11. A "command cell" refers to the locations accessed by the IOCR instruction (60, 61, 62, and 63 hex).

12. For I/0 channels, IOBUS bits 19-23 will be logical 1's for broadcast operations and these must be decoded before responding. It is not sufficient to simply decode bit 19 to determine if a broadcast operation is occurring. No X-Acknowledge or X-Resume signals shall be generated by the I/0 for broadcast operations.

13. Bit 0 (MSB) of all software instructions sent to the I/0 module, which are executed from an input chain program, will be forced to a logic 0 value." For example: The FBxX (Store Status) instruction would be received by the I/0 module as 7Bxx if the instruction was executed out of an input chain program.

PROCESSOR TO I/O CHANNEL IOBUS ACTIVITY - SOFTWARE I/O COMMANDS

SOFTWARE COMMAND MNEMONIC	HEX CODE	IOBUS ACTIVITY	HARDWARE ACTIONS
ACR 0 CCR 0,0	E000	A. Bit 16 = 1 Bit 17 = 1 Bit 18 = 0 Bit 19 = 1 Bit 20-23 = F XC = E0 x 0 if from command cell or output chain = 60 x 0 if from input chain XO = xxxx	A. Perform a master clear of the I/O channel hardware. Do <u>not</u> clear the I/O channel hardware image of CM-8 (control memory location 8) through CM-F (if these exist). NOTE: Bits 00 and 12 of XC may be ignored.
ACR 4 CCR 0,4	E004	A. Bit 16 = 1 Bit 17 = 1 Bit 18 = 0 Bit 19 = 1 Bit 20-23 = F XC = E0 x 4 if from command cell or output chain = 60 x 4 if from input chain XO = xxxx	A. Set the EIE (enable) flip-flop (if it exists). NOTE: Bits 00 and 12 of XC may be ignored.
ACR 5 CCR 0,5	E005	A. Bit 16 = 1 Bit 17 = 1 Bit 18 = 0 Bit 19 = 1 Bit 20-23 = F XC = E0 x 5 if from command cell or output chain = 60 x 5 if from input chain XO = xxxx	A. Clear the EIE (enable) flip-flop (if it exists). NOTE: Bits 00 and 12 of XC may be ignored.

PROCESSOR TO I/O CHANNEL IOBUS ACTIVITY - SOFTWARE I/O COMMANDS (Cont.)

SOFTWARE COMMAND MNEMONIC	HEX CODE	IOBUS ACTIVITY	HARDWARE ACTIONS
CCR a,12	E0aC	A. Bit 16 = 1 Bit 17 = 1 Bit 18 = 0 Bit 19 = 0 Bit 20-23 = Priority Number XC = E0 x C if from command cell or output chain = 60 x C if from input chain X0 = xxxx	A. Accept data word (X0). Set the EIE (enable) flip-flop (if it exists). NOTE: Bits 00 and 12 of XC may be ignored.
CCR a,13	E0aD	A. Bit 16 = 1 Bit 17 = 1 Bit 18 = 0 Bit 19 = 0 Bit 20-23 = Priority Number XC = E0 x D if from command cell or output chain = 60 x D if from input chain X0 = xxxx	A. Accept data word (X0). Clear the EIE (enable) flip-flop (if it exists). NOTE: Bits 00 and 12 of XC may be ignored.
CCR a,14	E0aE	A. Bit 16 = 1 Bit 17 = 1 Bit 18 = 0 Bit 19 = 0 Bit 20-23 = Priority Number XC = E0 x E if from command cell or output chain = 60 x E if from input chain X0 = xxxx	A. Accept data word (X0). Set the "Class III Mask" flip-flop. NOTE: Setting or enabling the "Class III Mask" flip-flop should allow EII, OCI, ICI to be sent in as events. Bits 00 and 12 of XC may be ignored.

t

PROCESSOR TO I/O CHANNEL IOBUS ACTIVITY - SOFTWARE I/O COMMANDS (Cont.)

SOFTWARE COMMAND MNEMONIC	HEX CODE	IOBUS ACTIVITY	HARDWARE ACTIONS
CCR a,15	E0aF	A. Bit 16 = 1 Bit 17 = 1 Bit 18 = 0 Bit 19 = 0 Bit 20-23 = Priority Number XC = E0 x F if from command cell or output chain = 60 x F if from input chain XO = xxxx	A. Accept data word (XO). Clear the "Class III Mask" flip-flop. NOTE: Clearing or disabling the "Class III Mask" flip-flop should drop the EII, OCI, ICI events, if they are present, and should prevent them from being sent in as events until the flip-flop is set. Bits 00 and 12 of XC may be ignored.
ICK a,y (from command cell)	E6a2 yyyy	A. Bit 16 = 1 Bit 17 = 1 Bit 18 = 0 Bit 19 = 0 Bit 20-23 = Priority Number XC = E6 x 2 XO = yyyy	A. Accept data word (XO). Set the ICR event. NOTE: See also E60m
OCK a,y (from command cell)	E6a6 yyyy	A. Bit 16 = 1 Bit 17 = 1 Bit 18 = 0 Bit 19 = 0 Bit 20-23 = Priority Number XC = E6 x 6 XO = yyyy	A. Accept data word (XO). Set the OCR event. NOTE: See also E60m

PROCESSOR TO I/O CHANNEL IOBUS ACTIVITY - SOFTWARE I/O COMMANDS (Cont.)

SOFTWARE COMMAND MNEMONIC	HEX CODE	IOBUS ACTIVITY	HARDWARE ACTIONS
WIM a,y,m (from command cell)	E7am yyyy	A. <u>If</u> m = 8 through F <u>then</u> Bit 16 = 1 Bit 17 = 1 Bit 18 = 0 Bit 19 = 0 Bit 20-23 = Priority Number XC = E7 x m XO = (yyyy) <u>Else</u> no IOBUS activity	A. Accept data word (XO). See notes for Hex Code E60m.
RIM a,y,m (from command cell)	EBam yyyy	A. No IOBUS activity	A. NA
SICR a,m (from command cell)	F8am	A. Bit 16 = 1 Bit 17 = 1 Bit 18 = 0 Bit 19 = 0 Bit 20-23 = Priority Number XC = F8 x m XO = xxxx	A. Accept data word (XO) Set of clear the disc by "m". NOTE: The discrete f determined by particular cha
SST a,y,m (from command cell)	FBam yyyy	A. Bit 16 = 1 Bit 17 = 0 Bit 18 = 0 Bit 19 = 0 Bit 20-23 = Priority Number XC = FB x m XI = Status Word	A. Return the Status Wor "m". NOTE: The definition Words and thei are determined the particular except that on the Status Wor MAP event (See

PROCESSOR TO I/O CHANNEL IOBUS ACTIVITY - SOFTWARE I/O COMMANDS (Cont.)

SOFTWARE COMMAND MNEMONIC	HEX CODE	IOBUS ACTIVITY	HARDWARE ACTIONS
IM a,y,m (cont.)	E2am yyyy	B. Bit 16 = 1 Bit 17 = 1 Bit 18 = 0 Bit 19 = 0 Bit 20-23 = Priority Number XC = E2ax if from output chain XC = 62ax if from input chain XO = xxxx	B. Accept data word (XO). If XC = E2ax, drop OCR. If XC = 62ax, drop ICR. Initiate transfer as specified by the "a" field, using IDR for input and ODR for output. NOTE: Typically IDR or ODR event is used to transfer data. When transfer is complete, ICR or OCR is raised again. This instruction may be implemented ignoring bit 07 of XC, thus giving the same hardware action as E3a0. CM-B will be loaded with the contents of the main memory address specified by CM-5 (BAP). Thus this command can be used to initiate input only if the contents of CM-B are of no concern; i.e., only if no output is going on at the same time.
IO a,y (from chain)	E3a0 yyyy	A. Bit 16 = 1 Bit 17 = 1 Bit 18 = 0 Bit 19 = 0 Bit 20-23 = Priority Number XC = E3ax if from output chain XC = 63ax if from input chain XO = xxxx	A. Accept data word (XO). If XC = E3ax, drop OCR. If XC = 63ax, drop ICR. Initiate transfer as specified by the "a" field, using IDR for input and ODR for output. NOTE: Typically IDR or ODR event is used to transfer data. When transfer is complete, ICR or OCR is raised again. This instruction may be implemented ignoring bit 07 of XC, thus giving the same hardware action as E2am.

SOFTWARE COMMAND MNEMONIC	HEX CODE	IOBUS ACTIVITY	HARDWARE ACTIONS
IO a,y (Cont.)			If a ≠ xx00, CM-B is loaded with the contents of the main memory address specified by CM-5 (BAP). Thus a = xx00 cannot be used to initiate output and a ≠ xx00 may be used for input only if the contents of CM-B are of no concern; i.e., only if no output of data/functions is going on at the same time. If a ≠ xx00 and this command is executed from an input chain, CM-6 is loaded with the updated contents of CM-2. Similarly, if a = xx00 and this command is executed from an output chain, CM-2 is loaded with the updated contents of CM-6. Thus, care must be exercised when using this command with channels that implement both input and output chains.
LCMK m,y (from chain)	E60m yyyy	A. <u>If</u> m = 2, 6, or 8 through F, <u>then</u> Bit 16 = 1 Bit 17 = 1 Bit 18 = 0 Bit 19 = 0 Bit 20-23 = Priority Number XC = E6xm if from output chain XC = 66xm if from input chain XO = yyyy. <u>Else</u> no IOBUS activity.	A. Accept data (XO). If m = 6, set OCR event (see E6a6). If m = 2, set ICR event (see E6a2). NOTE: CM-2 and CM-6 are the CAPs for the channel. In general, other values of "m" and the hardware usage of yyyy are determined by the needs of the particular channel design. It must be remembered, however, that the IOP does not support CM-C through CM-F and that CM-B is reserved for special use

PROCESSOR TO I/O CHANNEL IOBUS ACTIVITY - SOFTWARE I/O COMMANDS (Cont.)

SOFTWARE COMMAND MNEMONIC	HEX CODE	IOBUS ACTIVITY	HARDWARE ACTIONS
			associated with output data transfers (see Table A-3, ODR). Bit 07 of XC may be ignored so that the same hardware action occurs for E7am and E70m.
LCM m,y (from chain)	E70m yyyy	A. <u>If</u> m = 8 through F, <u>then</u> Bit 16 = 1 Bit 17 = 1 Bit 18 = 0 Bit 19 = 0 Bit 20-23 = Priority Number XC = E7xm if from output chain = 67xm if from input chain XO = (yyyy). <u>Else</u> no IOBUS activity.	A. Accept data (XO). See hardware actions for Hex Code E60m
SCM m,y	EB0m	No IOBUS activity	NA
HCR	EC00	A. <u>If</u> input chain, <u>then</u> Bit 16 = 1 Bit 17 = 1 Bit 18 = 0 Bit 19 = 0 Bit 20-23 = Priority Number XC = 6Cax (a is even) XO = xxxx and done. <u>Else</u> go to step B.	A. Accept data word (XO). Drop ICR.

PROCESSOR TO I/O CHANNEL IOBUS ACTIVITY – SOFTWARE I/O COMMANDS (Cont.)

SOFTWARE COMMAND MNEMONIC	HEX CODE	IOBUS ACTIVITY	HARDWARE ACTIONS
HCR (cont.)		B. Bit 16 = 1 Bit 17 = 1 Bit 18 = 0 Bit 19 = 0 Bit 20-23 = Priority Number XC = ECax (a is even) XO = xxxx	B. Accept data word (XO). Drop OCR.
IPR (from chain)	EC10	A. <u>If</u> input chain, <u>then</u> Bit 16 = 1 Bit 17 = 1 Bit 18 = 0 Bit 19 = 0 Bit 20-23 = Priority Number XC = 6Cax (a is odd) XO = xxxx and done. <u>Else</u> go to step B.	A. Accept data word (XO). Generate ICI event.
		B. Bit 16 = 1 Bit 17 = 1 Bit 18 = 0 Bit 19 = 0 Bit 20-23 = Priority Number XC = ECax (a is odd) XO = xxxx	B. Accept data word (XO). Generate OCI event.
ZF y (from chain)	EF00 yyyy	No IOBUS Activity	NA
SF y	EF10 yyyy	No IOBUS Activity	NA

PROCESSOR TO I/O CHANNEL IOBUS ACTIVITY - SOFTWARE I/O COMMANDS (Cont.)

SOFTWARE COMMAND MNEMONIC	HEX CODE	IOBUS ACTIVITY	HARDWARE ACTIONS
SJMC 0,y (from chain)	F200 yyyy	A. Bit 16 = 1 Bit 17 = 0 Bit 18 = 0 Bit 19 = 0 Bit 20-23 = Priority Number XC = F208 if from output chain = 7208 if from input chain XI = Status Word	A. Return Status Word (XI) in the format defined in 14112100 - Input/Output Subsystem Equipment Specification, with MSB = logical "1".
SJMC a,y (from chain) a ≠ 0	F2a0 yyyy	A. Bit 16 = 1 Bit 17 = 0 Bit 18 = 0 Bit 19 = 0 Bit 20-23 = Priority Number XO = F2a8 if from output chain = 72a8 if from input chain XI = Status Word	A. Return Status Word (XI) in the format defined in 14112100 - Input/Output Subsystem Equipment Specification. The MSB is to be cleared or set depending on the I/O channel condition specified by "a". (The software will do a jump if MSB = logical "1".) NOTE: The condition to be tested for a ≠ 0 are determined by the needs of the particular I/O channel design.
SFSC m (from chain)	F40m	A. Bit 16 = 1 Bit 17 = 1 Bit 18 = 0 Bit 19 = 0 Bit 20-23 = Priority Number XC = F4xm if from output chain = 74xm if from input chain XO = xxxx	A. Accept data word (XO). Perform the function as specified by "m". NOTE: The functions to be performed are determined by the needs of the particular I/O channel design.

PROCESSOR TO I/O CHANNEL IOBUS ACTIVITY - SOFTWARE I/O COMMANDS (Cont.)

SOFTWARE COMMAND MNEMONIC	HEX CODE	IOBUS ACTIVITY	HARDWARE ACTIONS
CSIR m (from chain)	F80m	A. Bit 16 = 1 Bit 17 = 1 Bit 18 = 0 Bit 19 = 0 Bit 20-23 = Priority Number XC = F8xm if from output chain = 78xm if from input chain XO = xxxx	A. Accept data word (XO). Set or clear the discrete as specified by "m". NOTE: The discrete is as determined by the needs of the particular channel design.
CSST y,m (from chain)	FB0m yyyy	A. Bit 16 = 1 Bit 17 = 0 Bit 18 = 0 Bit 19 = 0 Bit 20-23 = Priority Number XC = F8xm if from output chain = 78xm if from input chain XI = Status Word	A. Return the status word as specified by "m". NOTE: The definitions of the Status Words and their dependence on "m" are determined by the needs of the particular channel design except that one status word must be the status word for the MAP event (see Table A-3).
BJ m,y (from chain)	FD0m yyyy	No IOBUS Activity	NA
XCM m,y (from chain)	FE0m yyyy	A. <u>If</u> m = 8 through F, <u>then</u> Bit 16 = 1 Bit 17 = 1 Bit 18 = 0 Bit 19 = 0 Bit 20-23 = Priority Number XC = E6xM if from output chain = 66xM if from input chain XI = (yyyy*). <u>Else</u> no IOBUS activity.	A. Accept data word (XO). NOTE: The use of this data word is dependent on the needs of the particular channel design. Refer to note for Hex Code E60m. (yyyy*) is the contents of memory location (y, logically ORed with 1).

The second section of Appendix D describes XBUS interaction between the processor and I/O module in response to a particular raised event. All events and their associated XBUS activity are presented

Notes:

1. For I/0 channels, bits 17-23 of the IOBUS are defined only during the CONTROL portion of the cycle. Bit 16 is as shown in the table during the CONTROL time, and is driven to a logic 0 by the I/0 module during the DATA portion of the IOBUS cycle

2. For I/0 channels, IOBUS bit 18 is always a logic 0. It must be decoded.

3. For I/0 channels, IOBUS bit 19 is a logic 0 for all non-"BROADCAST" operations.

4. On the IOBUS, complement polarity is used so that a logical 1 is represented by a ground potential.

5. The values shown in this table are logic true values.

6. An "x" in the value for XC or XO implies that those bits are indeterminate and the I/0 hardware shall not decode them.

7. The hardware bit numbering scheme is used in this table (bit 0 = MSB).

8. The comment "Accept data word (XO)" simply implies that the I/0 channel must respond with X-Acknowledge and X-Resume signals. The data is not necessarily used.

9 For I/O channels, XBUS bits 20-23 contain the priority number of the channel.

10. A “K” represents the priority number of the channel that generated the event. “K*E” represents the priority number obtained by forcing the LSB of K to a zero.

11. Unless otherwise stated, all XBUS activity occurs for the channel whose event is being serviced.

12. “BCW*” is bits 04-15 of CM-0 or CM-4

13. “CM-n” is Control Memory word n.

PROCESSOR TO I/O CHANNEL IOBUS ACTIVITY - EVENT RESPONSES

CLASS	EVENT DISK	NAME	FIRMWARE	CHANNEL HARDWARE ACTION
001	0/4	RT CMD	A. IOBUS Activity Bit 16 = 1 Bit 17 = 0 XC = EFx0 XI = Index Status Word	A. Return Index Status Word (XI) and clear RT CMD event.
			B. Load CM-5 (BAP) with the contents of main memory address (ATP + T/R/Subaddress); where ATP is the content of CM-7 (ATP) and T/R is bit 10 of Index Status Word and Subaddress is bits 11-15 of Index Status Word.	
			C. <u>If</u> T/R = 0 or Subaddress = 0, <u>then</u> done; <u>else</u> go to Step D.	
			D. IOBUS Activity Bit 16 = 0 Bit 17 = 1 XC = EFx1 XO = Contents of main memory address, specified by CM-5.	D. Accept data word (XO) and clear ODR. NOTE: This data word (XO) is the first data word. The remaining data words will be sent in response to an ODR. So if ODR is going to be raised for the first data word, it should be raised simultaneously with RT CMD so that it is cleared at this time.
			E. Add 1 to contents of CM-5 (BAP).	
			F. Load CM-B with contents of main memory address specified by CM-5.	
001	1/5	ODR	A. IOBUS Activity Bit 16 = 0 Bit 17 = 1 XC = EFx1 XC = Contents of CM-B (Data Word)	A. Accept data word (XO) and clear ODR.

PROCESSOR TO I/O CHANNEL IOBUS ACTIVITY – EVENT RESPONSES (Cont.)

CLASS	EVENT DISK	NAME	FIRMWARE	CHANNEL HARDWARE ACTION
001 (cont.)	1/5	ODR	B. Add 1 to contents of CM-5 (BAP).	
			C. Load CM-B with contents of main memory address specified in CM-5.	
001	2/6	IDR	A. IOBUS Activity Bit 16 = 0 Bit 17 = 0 XC = EFx2 XI = Data Word	A. Return Data Word (XI) and clear IDR.
			B. Store Data Word (XI) in main memory at the address specified by CM-5 (BAP).	
			C. Add 1 to contents of CM-5.	
001	3/7	—	A. Set BIT indicator.	—
010	0/4	UCR	A. IOBUS Activity Bit 16 = 1 Bit 17 = 0 XC = EFx4 XI = Unique Channel Control Word	A. Return Unique Channel Control Word (XI) and Clear UCR.
			B. 1. <u>If</u> XI = xx00, <u>then</u>	
			a. IOBUS Activity (Channel K•E) Bit 16 = 0 Bit 17 = 0 XC = EFx7 XI = Data Word K•E	a. Return Data Word K•E NOTE: EFx7 is response to IDR, also.

PROCESSOR TO I/O CHANNEL IOBUS ACTIVITY - EVENT RESPONSES (Cont.)

CLASS	EVENT DISK	NAME	FIRMWARE	CHANNEL HARDWARE ACTION
010 (cont.)	0/4	UCR	b. IOBUS Activity (Channel K) Bit 16 = 0 Bit 17 = 0 XC = EFx7 XI = Data Word K	b. Return Data Word K NOTE: EFx7 is also the response to IDR.
			c. Store Data Word K•E in main memory at the address = Data Word K.	
			d. Store Data Word K in main memory at the address = Data Word K + 1.	
			e. Decrement CM-0 (BCW*) for Channel K.	
			f. <u>If</u> BCW* = 0, <u>then</u> IOBUS Activity (Channel K): Bit 16 = 1 Bit 17 = 1 XC = EFxA XO = xxxx <u>Else</u> go to step B1g.	f. Accept data. NOTE: EFxA is also the response to IDR when BCW* = 0.
			g. Done.	
			<u>Else</u> go to step B2.	
			2. <u>If</u> XI = xx04, <u>then</u>	
			a. IOBUS Activity (Channel K) Bit 16 = 0 Bit 17 = 0 XC = EFx2 XI = Data Word K	a. Return Data Word K. NOTE: EFx2 is also the response to IDR.
			b. Read main memory using address = Data Word K.	

PROCESSOR TO I/O CHANNEL IOBUS ACTIVITY - EVENT RESPONSES (Cont.)

CLASS	EVENT DISK	NAME	FIRMWARE	CHANNEL HARDWARE ACTION
010 (cont.)	0/4	UCR	c. IOBUS Activity (Channel K•E) Bit 16 = 0 Bit 17 = 1 XC = EFx6 XO = Results of previous step.	c. Accept data. NOTE: EFx6 is also the response to ODR/EFR.
			d. Read main memory with address = Data Word K + 1.	
			e. IOBUS Activity (Channel K) Bit 16 = 0 Bit 17 = 1 XC = EFx6 XO = Results of previous step.	e. Accept data. NOTE: EFx6 is also the response to ODR/EFR.
			f. Decrement the contents of CM-4 (BCW*) for (Ch. K).	
			g. <u>If</u> CM-4 (BCW*) = 0, <u>then</u> IOBUS Activity Bit 16 = 1 Bit 17 = 1 XC = EFx9 XO = xxxx; and done. <u>Else</u>, go to step B2h.	g. Accept data. NOTE: EFx9 is also the response to ODR/EFR when CM-4 (BCW*) = 0.
			h. Done.	
			<u>Else</u>, go to step B3.	
			3. <u>If</u> XI = xx10, <u>then</u>	
			a. IOBUS Activity (Channel K•E) Bit 16 = 0 Bit 17 = 0 XC = EFx5 XI = Data Word K•E	a. Return data word K•E. NOTE: EFx5 is also the response to EIR.

PROCESSOR TO I/O CHANNEL IOBUS ACTIVITY – EVENT RESPONSES (Cont.)

CLASS	EVENT DISK	NAME	FIRMWARE	CHANNEL HARDWARE ACTION
010 (cont.)	0/4	UCR	b. Store Data Word K•E at main memory address = (0080 + <u>Logical</u> Ch. No. –1 for Priority Number K).	
			c. Done.	
			<u>Else</u>, go to step B4	
			4. <u>If</u> XI = xm14, <u>then</u>	
			a. IOBUS Activity (Channel K) Bit 16 = 0 Bit 17 = 0 XC = EFx7 XI = Data Word K	a. Return Data Word K. NOTE: EFx7 is also the response to IDR.
			b. Load CM specified by bits 4–7 of Unique Channel Control Word (for Ch. K) with Data Word K.	
			c. If bits 4–7 of Unique Channel Control Word = 5 (BAP), read main memory at address = Data Word K and load the data into CM–B (Ch. K).	
			d. Done.	
			<u>Else</u>, go to step B5.	
			5. <u>If</u> XI = xm18, <u>then</u>	
			a. Read CM location specified by bits 4–7 of Unique Channel Control Word.	

PROCESSOR TO I/O CHANNEL IOBUS ACTIVITY – EVENT RESPONSES (Cont.)

CLASS	EVENT DISK	NAME	FIRMWARE	CHANNEL HARDWARE ACTION
010 (cont.)	0/4	UCR	b. IOBUS Activity Bit 16 = 0 Bit 17 = 1 XC = EFx6 XO = Data from previous step.	b. Accept data. NOTE: EFx6 is also the response to ODR.
			c. Done.	
			<u>Else</u>, go to step B6.	
			6. <u>If</u> XI = xx1C, <u>then</u>	
			a. IOBUS Activity Bit 16 = 0 Bit 17 = 0 XC = EFx7 XI = Address Word	a. Return Address Word (XI) NOTE: EFx7 is also the response to IDR.
			b. IOBUS Activity Bit 16 = 0 Bit 17 = 0 XC = EFx7 XI = Data Word	b. Return Data Word (XI) NOTE: EFx7 is also the response to IDR.
			c. Store Data Word in main memory at address = Address Word.	
			d. Done.	
			<u>Else</u>, go to step B7.	
			7. <u>If</u> XI = xx20, <u>then</u>	
			a. IOBUS Activity Bit 16 = 0 Bit 17 = 0 XC = EFx7 XI = Address Word	a. Return Address Word (XI) NOTE: EFx7 is also the response to IDR.

CLASS	EVENT DISK	NAME	FIRMWARE	CHANNEL HARDWARE ACTION
010 (cont.)	0/4	UCR	b. Read main memory at address = Address Word	
			c. IOBUS Activity Bit 16 = 0 Bit 17 = 1 XC = EFx6 XO = Data read from main memory in previous step.	c. Accept data. NOTE: EFx6 is also the response to ODR.
			d. Done.	
			Else, go to step B8.	
			8. *If*, XI = xx24, *then*	
			a. IOBUS Activity Bit 16 = 0 Bit 17 = 0 XC = EFx7 XI = Address Word	a. Return Address Word (XI) NOTE: EFx7 is also the response to IDR.
			b. IOBUS Activity Bit 16 = 0 Bit 17 = 0 XC = EFx7 XI = Increment Word	b. Return Increment Word (XI) NOTE: EFx7 is also the response to IDR.
			c. Read main memory at address = Address Word.	
			d. Add Increment Word to the data from the previous step and store the result in main memory address = Address Word.	
			e. Done.	
			Else, go to step B9.	
			9. Results are indeterminate.	

PROCESSOR TO I/O CHANNEL IOBUS ACTIVITY – EVENT RESPONSES (Cont.)

CLASS	EVENT DISK	NAME	FIRMWARE	CHANNEL HARDWARE ACTION
010	1/5	EIR	A. IOBUS Activity Bit 16 = 0 Bit 17 = 0 XC = EFx5 XI = Interrupt Word	A. Return Interrupt Word. Clear EIR. NOTE: EIE may be cleared at this time if desired.
			B. Store the Interrupt Word in main memory at address = (0080 + Logical Channel No.)	
010	2/6	ODR	NOTE: TM = Bits 00 and 01 of CM-4 (BCW)	
			A. 1. <u>If</u> TM = 3, <u>then</u>	
			a. IOBUS Activity (Channel K•E) Bit 16 = 0 Bit 17 = 1 XC = EFx6 XO = Contents of CM-B (Ch. K) (Data Word)	a. Accept data word
			b. Add 1 to contents of CM-5 (BAP).	
			c. Read main memory at address = contents of CM-5.	
			d. IOBUS Activity (Channel K) Bit 16 = 0 Bit 17 = 1 XC = EFx6 XO = Data read in previous step.	d. Accept data word. Clear ODR.
			e. Go to step B.	
			<u>Else</u>, go to step A2.	

PROCESSOR TO I/O CHANNEL IOBUS ACTIVITY – EVENT RESPONSES (Cont.)

CLASS	EVENT DISK	NAME	FIRMWARE	CHANNEL HARDWARE ACTION
010 (cont.)	2/6	ODR	2. <u>If</u> TM = 0 or 2, <u>then</u>	
			a. IOBUS Activity (Channel K) Bit 16 = 0 Bit 17 = 1 XC = EFx6 XO = Contents of CM-B (Ch. K).	a. Accept data word. Clear ODR.
			b. Go to step B.	
			<u>Else</u>, go to step A3.	
			NOTE: "B" is bit 03 of CM-4 (BCW). "BCW" is bits 04-15 of CM-4 (BCW).	
			3. <u>If</u> TM = 1 <u>and</u> B = 0, <u>then</u>	
			a. IOBUS Activity (Channel K) Bit 16 = 0 Bit 17 = 1 XC = EFx6 XO = Contents of CM-B (Ch. K) most significant byte, right-justified, zero-filled.	a. Accept Data Word Clear ODR.
			b. Set B = 1	
			c. Decrement BCW*.	
			d. <u>If</u> BCW* = 0, <u>then</u> go to step D, <u>else</u> done.	
			<u>Else</u> go to step A4.	

PROCESSOR TO I/O CHANNEL IOBUS ACTIVITY - EVENT RESPONSES (Cont.)

CLASS	EVENT DISK	NAME	FIRMWARE	CHANNEL HARDWARE ACTION
010 (cont.)	2/6	ODR	4. <u>If</u> TM = 1 <u>and</u> B = 1, <u>then</u>	
			a. IOBUS Activity (Channel K) Bit 16 = 0 Bit 17 = 1 XC = EFx6 XO = Contents of CM-B (Ch. K) least significant byte, zero-filled.	a. Accept Data Word. Clear ODR.
			b. Set B = 0 and go to step B.	
			<u>Else</u>, go to step B.	
			B. Increment contents of CM-5 (BAP) (Ch. K).	
			C. Decrement contents of CM-4 (BCW*).	
			D. <u>If</u> BCW* = 0, <u>then</u> IOBUS Activity Bit 16 = 1 Bit 17 = 1 XC = EFx9 XO = xxxx and done; <u>Else</u>, go to step E.	D. Accept Data Word (XO). Set OCR. Clear ODR (if set).
			E. Read main memory at address = contents of CM-5 (BAP) and store the data in CM-B (Data).	
010	3/7	IDR	NOTES: TM = Bits 00 and 01 of CM-0 (BCW) (ch. K) B = Bit 03 of CM-0 (BCW) BCW* is bits 04-15 of CM-0 (BCW).	

PROCESSOR TO I/O CHANNEL IOBUS ACTIVITY - EVENT RESPONSES (Cont.)

CLASS	EVENT DISK	NAME	FIRMWARE	CHANNEL HARDWARE ACTION
010 (cont.)	3/7	IDR	A. <u>If</u> TM = 3, <u>then</u>	
			a. IOBUS Activity (Channel K•E) Bit 16 = 0 Bit 17 = 0 XC = EFx7 XI = Data Word	a. Return Data Word (XI). Clear IDR.
			b. Store Data Word from previous step in main memory at address = Contents of CM-1 (BAP) for Channel K.	
			c. Increment CM-1 (BAP) - Channel K	
			d. Go to step B.	
			<u>Else</u> go to step B.	
			B. IOBUS Activity (Ch. K) Bit 16 = 0 Bit 17 = 0 XC = EFx7 XI = Data Word	B. Return Data Word (XI). Clear IDR.
			C. <u>If</u> TM = 0 <u>then</u> go to step H, <u>else</u> go to step D.	
			D. <u>If</u> TM = 2 or 3, <u>then</u>	
			a. In main memory at address = contents of CM-1 (BAP); write the data obtained in step B.	
			b. Go to step G.	
			<u>Else</u> go to step E.	

PROCESSOR TO I/O CHANNEL IOBUS ACTIVITY - EVENT RESPONSES (Cont.)

CLASS	EVENT DISK	NAME	FIRMWARE	CHANNEL HARDWARE ACTION
010 (cont.)	3/7	IDR	E. _If_ TM = 1 and B = 0, _then_ a. Write the 8 LSB of the data obtained in step B to byte 0 in main memory at address = contents of CM-1 (BAP). b. Set B = 1 c. Go to step H. _Else_ go to step F.	
			F. _If_ TM = 1 and B = 1, _then_ a. Write the 8 LSB of the data obtained in step B to byte 1 in main memory at address = contents of CM-1 (BAP). b. Set B = 0 c. Go to step G. _Else_ go to step G.	
			G. Increment CM-1 (BAP).	
			H. Decrement CM-0 (BCW*).	
			I. _If_ BCW* = 0, _then_ IOBUS Activity Bit 16 = 1 Bit 17 = 1 XC = EFxA XO = xxxx _else_, done.	I. Accept Data Word (XO). Set ICR. Clear IDR.

PROCESSOR TO I/O CHANNEL IOBUS ACTIVITY - EVENT RESPONSES (Cont.)

CLASS	EVENT DISK	NAME	FIRMWARE	CHANNEL HARDWARE ACTION
100	0/4	MAP	A. IOBUS Activity Bit 16 = 1 Bit 17 = 0 XC = EFx8 XI = Status Word	A. Return Status Word. This word shall have the format as described in document 14112100 - Input/Output Subsystem Equipment Specification. (Note that the software bit-numbering scheme is used.) Clear MAP.
			B. Generate MAP Entry. (Correspondence between Logical Channel Number and Priority Number).	
100	1/5	OCR	A. Read 16 MSB of an instruction from main memory at address = contents of CM-6 (CAP).	
			B. Increment CM-6 (CAP).	
			C. <u>If</u> 2-word instruction, <u>then</u>	
			1. Read 16 LSB of instruction from main memory at address = contents of CM-6 (CAP).	
			2. Increment CM-6 and go to step D.	
			<u>Else</u> go to step D.	
			D. Execute the chain instruction.	D. Depends on the chain instruction. See table A-2.
100	2/6	ICR	A. Read 16 MSB of an instruction from main memory at address = contents of CM-2 (CAP).	
			B. Increment CM-2 (CAP).	

CLASS	EVENT DISK	NAME	FIRMWARE	CHANNEL HARDWARE ACTION
100 (cont.)	2/6	ICR	C. If 2-word instruction, then 1. Read 16 LSB of instruction from main memory at address = contents of CM-2 (CAP). 2. Increment CM-6 and go to step D. Else go to step D.	
			D. Clear MSB of instruction.	
			E. Execute the chain instruction.	D. Dependent on chain instruction. See table A-2.
100	3/7	EIR	A. IOBUS Activity Bit 16 = 1 Bit 17 = 0 XC = EFxB XI = Interrupt Word	A. Return Interrupt Word (XI). Clear EIR. NOTE: EIE may be cleared at this time if desired.
			B. Store Interrupt Word in main memory at address = 0080 + Logical Channel No.	
111	---	---	NOTE: All Class 111 events result in a software Class III interrupt. (See Section 9 of 14122000 - AN/AYK-14 Instruction Set Programmer's Reference Manual). If an IOP is in the system acting as a controller, the Class III interrupt is passed to the CPU. If the IOP is acting as a processor, the IOP handles the Class III interrupt. If there is no IOP, the CPU handles the Class III Interrupt. In any case, the result as seen by the I/O channel at the IOBUS interface is as described below.	

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX E: VHDL SOURCE CODE

<u>Hirearchy Of Souce Code</u>

Level I	Level II	Level III
Adapter_top.vhd	Sdramcnt.vhd	Xs_pckg.vhd
	Mbus_controller.vhd	Odd_parity.vhd
		Grant_logic.vhd
		Xs_pckg.vhd
	Xbus_controller.vhd	X_grant_logic.vhd
		Xs_pckg.vhd
	Add_select.vhd	
	xs_pckg.vhd	
	evtfsm.vhd	

==

Memory Arbitrator <Mem_Arbitrator.vhdl>

==

Project: AYK-14 VHSIC Processor Module Hardware Emulator
Component: Memory Use Arbitrator
Description: Sate Machine which provides a rotating access scheme to provide access to the on chip memory to all users, specifically the Processor, the Xbus, and the Mbus.

Author: LT Bryan Fetter, USN
Advisor: Dr. Russ Duren
Co-advisor: Dr. Hersch Loomis
Location: Naval Postgraduate School

Created: 30 August 2002
Modified: 6 November 2002
Simulated:
Target: XCV1000E FG1156
Software: Synplify Pro 7.1
Notes:

Disclaimer: NPS, makes no warranty for the use of this code or design. This code is provided "As Is". NPS, assumes no responsibility for any errors, which may appear in this code, nor does it make a commitment to update the information contained herein. NPS specifically disclaims any implied warranties of fitness for a particular purpose.

==

```
library IEEE;
use IEEE.std_logic_1164.all;

entity mem_arbitrator is

	generic(
		DATA_WIDTH: natural := 32;
		ADDR_WIDTH: natural := 23
		);
	port (
		Clk:	in	std_logic;
		RST:	in	std_logic;
		--Signals from SDRAM Controller
		Mem_Done:	in	std_logic;
			-- Memory Available signal from SDAM Ctr
```

```
            RD:     out     std_logic;
            WR:     out     std_logic;
            hAddr:  out     std_logic_vector(ADDR_WIDTH-1 downto 0);
            hData_In:       out     std_logic_vector(DATA_WIDTH-1 downto 0);
            --Out TO SDRAM
            hData_Out:      in      std_logic_vector(DATA_WIDTH-1 downto 0);
            --In FROM SDRAM
            --Signals from Processor
            P_Addr_In:      in      std_logic_vector(ADDR_WIDTH-1 downto 0);
            -- Memory Address In
            P_Data_In:      in      std_logic_vector(DATA_WIDTH-1 downto 0);
            P_Data_Out:     out     std_logic_vector(DATA_WIDTH-1 downto 0);
            P_Mem_Done:             out     std_logic;
            P_RD: in        std_logic;
            P_WR:in         std_logic;
            --Signals from MBus
            M_Addr_In:      in      std_logic_vector(ADDR_WIDTH-1 downto 0);
            -- Memory Address In
            M_Data_In:      in      std_logic_vector(DATA_WIDTH-1 downto 0);
            M_Data_Out:     out     std_logic_vector(DATA_WIDTH-1 downto 0);
            M_Mem_Done:             out     std_logic;
            M_RD:           in      std_logic;
            M_WR:           in      std_logic;
            --Signals from XBus
            X_Addr_In:      in      std_logic_vector(ADDR_WIDTH-1 downto 0);
            -- Memory Address In
            X_Data_In:      in      std_logic_vector(DATA_WIDTH-1 downto 0);
            X_Data_Out:     out     std_logic_vector(DATA_WIDTH-1 downto 0);
            X_Mem_Done:             out     std_logic;
            X_RD:in         std_logic;
            X_WR:           in      std_logic

            );

end mem_arbitrator;

architecture behavioral of mem_arbitrator is

constant        Addr_Z:         Std_Logic_Vector(ADDR_WIDTH-1
                                downto 0):="ZZZZZZZZZZZZZZZZZZZZZZ";

type statetype is (
            Idle,   -- Idle state when no entity is requesting Memory
            P,              -- State when Processor has control of Memory
            X,              -- State when Xbus has control of Memory
            M               -- State when Mbus has control of Memory
```

```
);

signal curr_state, next_state : statetype ;
signal P_REQ,M_REQ,X_REQ : std_logic;

begin

P_REQ <= P_RD or P_WR;
M_REQ <= M_RD or M_WR;
X_REQ <= X_RD or X_WR;

--Process to determine next state

nxtStProc:
 process        (P_REQ,M_REQ,X_REQ,curr_state,Mem_Done,P_RD,P_WR,
                 M_RD,M_WR,X_RD,X_WR,next_state)

begin

  case curr_state is
    when Idle =>
      if P_REQ = '1' then          --First If statements determine if any user wants memory
        next_state <= P;
      elsif X_REQ = '1' then
        next_state <= X;
      elsif M_REQ = '1' then
        next_state <= M;
      else
        next_state <= Idle;
      end if;

      case next_state is
--As soon as the highest priority user is determined from statements above, the RD or
WR signal is sent to the SDRAM controller

        when Idle =>
          RD <= '0';
--This is to ensure that the SDRAM controler goes to the RW state on the following clock
          WR <= '0';
        when P =>
          RD <= P_RD;
          WR <= P_WR;
        when X =>
          RD <= X_RD;
          WR <= X_WR;
        when M =>
```

```
        RD <= M_RD;
        WR <= M_WR;
      when others =>
        null;
      end case;

  when P =>

    if Mem_Done = '0' then
      --Each state remains in that state until the Mem_Done signal indicates that memory is available
      RD <= P_RD;
      WR <= P_WR;
      next_state <= P;
    elsif Mem_Done = '1' then
--The next state priority is determined by the order of the if statements
      if X_REQ = '1' then
        next_state <= X;
      elsif M_REQ = '1' then
        next_state <= M;
      else
--If the same user is the only one that wants memory, the state must first go to the Idle state.This is to prevent timing issues in regrard to reasserting the Request signals. This may not be needed after testing with hardware
        next_state <= Idle;
      end if;
    end if;                              .

  when M =>

    if Mem_Done = '0' then
      RD <= M_RD;
      WR <= M_WR;
      next_state <= M;
    elsif Mem_Done = '1' then
      if P_REQ = '1' then
        next_state <= P;
      elsif X_REQ = '1' then
        next_state <= X;
      else
        next_state <= Idle;
      end if;
    end if;

  when X =>
```

```
    if Mem_Done = '0' then
      RD <= X_RD;
      WR <= X_WR;
      next_state <= X;
    elsif Mem_Done = '1' then
      if M_REQ = '1' then
        next_state <= M;
      elsif P_REQ = '1' then
        next_state <= P;
      else
        next_state <= Idle;
      end if;
    end if;

  when others =>
    null;

  end case;

end process nxtStProc;

--This process determines the output signals based on the current state and input signals

outConProc:
process(curr_state,next_state,P_RD,P_WR,M_RD,M_WR,X_RD,X_WR,X_Addr_In,
        P_Addr_In,M_Data_In,hData_Out,P_Data_In,X_Data_In,M_Addr_In,
        Mem_Done)

begin

 case curr_state is

  when Idle =>
        --In Idle, all the memory done signals are set to '0' to prevent misreading of
invalid memory signals
        X_Mem_Done <= '0';
        P_Mem_Done <= '0';
        M_Mem_Done <= '0';
        --hAddr <= ADDR_Z;          --Connect Address bus to high Z

  when P =>
        hAddr <= P_Addr_In;         --Connect P lines to Input/Output Lines
        P_Data_Out <= hData_Out;
        hData_In <= P_Data_In;
        P_Mem_Done <= Mem_Done;
```

```
   when X =>
         hAddr <= X_Addr_In;               --Connect X lines to Input/Output Lines
         X_Data_Out <= hData_Out;
         hData_In <= X_Data_In;
         X_Mem_Done <= Mem_Done;

   when M =>
         hAddr <= M_Addr_In;               --Connect M lines to Input/Output Lines
         M_Data_Out <= hData_Out;
         hData_In <= M_Data_In;
         M_Mem_Done <= Mem_Done;

   when others =>
         hAddr <= ADDR_Z;                  --Connect Address bus to high Z

end case;

end process ;

--Process to go from state to state (syncronize outputs)

state_to_state: process (CLK,RST)
         --Procedes to next state when Memory Operation is done
begin
         if       (RST = '1') then
                  curr_state <= Idle;
         elsif   (CLK'EVENT and CLK='1' ) then --and Mem_Done = '1') then
                  curr_state <= next_state;
         end if;


end process;


end behavioral;
```

==

Address Selector <Add_Select.vhd>

==

Project: AYK-14 VHSIC Processor Module Hardware Emulator
Component: Address Selector for MBUS
Description: Address multiplexor that provides the Desire signals to the MBUS ARbitrator for requests for memory from the Processor that are out of range of the On Board Memory. It defaults values to High Z when the data requested is available on board.

Author: LT Bryan Fetter, USN
Advisor: Dr. Russ Duren
Co-advisor: Dr. Hersch Loomis
Location: Naval Postgraduate School

Created: 25 October 2002
Modified: 7 November 2002
Simulated:
Target: XCV1000E FG1156
Software: Foundation 4.2i
Notes:

==

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

package Add_Select is

component Add_Select
   port (
      --Processor Side
      Add_In_Proc: in unsigned (22 downto 0);
      Data_WR_Proc: in unsigned (31 downto 0);
      Data_RD_Proc: out unsigned (31 downto 0);
```

```
        RD_Req_in_Proc: in STD_LOGIC;
        WR_Req_in_Proc: in STD_LOGIC;
        Mem_req_Done_Proc: out STD_LOGIC;
      --Mem_Writedoub_request: in STD_LOGIC;
      --IR_Bus: in unsigned ( 31 downto 0);
      --Protect: in unsigned (2 downto 0);

        --MBUS Side
        Data_RD_MBUS: in unsigned (31 downto 0);
        Data_WR_MBUS: out unsigned (31 downto 0);
        Add_out_MBUS: out unsigned (22 downto 0);
        RD_Req_out_MBUS: out STD_LOGIC;
        WR_Req_out_MBUS: out STD_LOGIC;
        Proc_Desire_L_MBUS: out STD_LOGIC;
        Mem_req_Done_MBUS: in STD_LOGIC;

        --OBM Side
        Add_In_OBM: out unsigned (22 downto 0);
        Data_RD_OBM: in unsigned (31 downto 0);
        Data_WR_OBM: out unsigned (31 downto 0);
        RD_Req_OBM: out STD_LOGIC;
        WR_Req_OBM: out STD_LOGIC;
        Mem_req_Done_OBM: in STD_LOGIC
      --Mem_Writedoub_request_OBM: out STD_LOGIC;
      --IR_Bus_OBM: out unsigned ( 31 downto 0);
      --Protect_OBM: out unsigned (2 downto 0);
    );
end component;

end package Add_Select;


library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity Add_Select is
    port (
        --Processor Side
        Add_In_Proc: in unsigned (22 downto 0);
        Data_WR_Proc: in unsigned (31 downto 0);
        Data_RD_Proc: out unsigned (31 downto 0);
        RD_Req_in_Proc: in STD_LOGIC;
        WR_Req_in_Proc: in STD_LOGIC;
        Mem_req_Done_Proc: out STD_LOGIC;
      --Mem_Writedoub_request: in STD_LOGIC;
```

```
    --IR_Bus: in unsigned ( 31 downto 0);
    --Protect: in unsigned (2 downto 0);

     --MBUS Side
     Data_RD_MBUS: in unsigned (31 downto 0);
     Data_WR_MBUS: out unsigned (31 downto 0);
     Add_out_MBUS: out unsigned (22 downto 0);
     RD_Req_out_MBUS: out STD_LOGIC;
     WR_Req_out_MBUS: out STD_LOGIC;
     Proc_Desire_L_MBUS: out STD_LOGIC;
     Mem_req_Done_MBUS: in STD_LOGIC;

     --OBM Side
     Add_In_OBM: out unsigned (22 downto 0);
     Data_RD_OBM: in unsigned (31 downto 0);
     Data_WR_OBM: out unsigned (31 downto 0);
     RD_Req_OBM: out STD_LOGIC;
     WR_Req_OBM: out STD_LOGIC;
     Mem_req_Done_OBM: in STD_LOGIC
    --Mem_Writedoub_request_OBM: out STD_LOGIC;
    --IR_Bus_OBM: out unsigned ( 31 downto 0);
    );
end Add_Select;

architecture Add_Select_arch of Add_Select is

constant Mem_Blk_1_L : natural := 1048576 ;
--Lower bound of VPM Master OBM (100000H)
constant Mem_Blk_1_H : natural := 2097151 ;
--Upper bound of VPM Master OBM (1FFFFFH)
constant Mem_Blk_2_L : natural := 2097152 ;
--Lower bound of VPM Slave1 OBM (200000H)
constant Mem_Blk_2_H : natural := 3145727 ;
--Upper bound of VPM Slave1 OBM (2FFFFFH)

signal Address : unsigned (22 downto 0);
--signal Data_RD : unsigned (31 downto 0);
signal Data_RD_M : unsigned (31 downto 0);
signal Data_RD_O : unsigned (31 downto 0);
signal Data_WR: unsigned (31 downto 0);
signal RD_Req : std_logic;
signal WR_Req : std_logic;
signal Mem_req_Done : std_logic;

begin
```

```
Address <= Add_In_Proc;
--Data_RD_Proc <= Data_RD;
Data_RD_M <= Data_RD_MBUS;
Data_RD_O <= Data_RD_OBM;
Data_WR <= Data_WR_Proc;
RD_Req <= RD_Req_In_Proc;
WR_Req <= WR_Req_In_Proc;
Mem_req_Done_Proc <= Mem_req_Done;


process
(Address,Data_WR,RD_Req,WR_Req,Data_RD_MBUS,Mem_req_Done_MBUS,
        Data_RD_OBM,Mem_req_Done_OBM,Data_RD_M,Data_RD_O)

  begin
    --If address is in OBM range, conect signals to OBM
    --and put MBUS signals to High Z
    if (Address >= to_unsigned(Mem_Blk_1_L,23)
        and Address <= to_unsigned(Mem_Blk_1_H,23)) then
      --Connect Signal to OBM
      Add_In_OBM <= Address;
      Data_WR_OBM <= Data_WR;
      Data_RD_Proc <= Data_RD_O;
      RD_Req_OBM <= RD_Req;
      WR_Req_OBM <= WR_Req;
      Mem_req_Done <= Mem_req_Done_OBM;
      --High Z signals to MBUS
      Add_out_MBUS <= (others => 'Z');
      Data_WR_MBUS <= (others => 'Z');
      RD_Req_out_MBUS <= '0';
      WR_Req_out_MBUS <= '0';
      Proc_Desire_L_MBUS <= '1';


    --If address is out of OBM range, connect signals to MBUS
    --and put OBM signals High Z
    elsif (Address < to_unsigned(Mem_Blk_1_L,23)
          or (Address >= to_unsigned(Mem_Blk_2_L,23)
          and Address <= to_unsigned(Mem_Blk_2_H,23))) then
      --Connect signals to MBUS
      Add_out_MBUS <= Address;
      Data_WR_MBUS <= Data_WR;
      Data_RD_Proc <= Data_RD_M;
      RD_Req_out_MBUS <= RD_Req;
      WR_Req_out_MBUS <= WR_Req;
      Mem_req_Done <= Mem_req_Done_MBUS;
```

```
    Proc_Desire_L_MBUS <= (RD_Req NOR WR_Req);
    --High Z signals to OBM
    Add_In_OBM <= (others => 'Z');
    Data_WR_OBM <= (others => 'Z');
    RD_Req_OBM <= '0';
    WR_Req_OBM <= '0';


  else
    Data_RD_Proc <= (others => 'Z');
    Add_out_MBUS <= (others => 'Z');
    Data_WR_MBUS <= (others => 'Z');
    RD_Req_out_MBUS <= '0';
    WR_Req_out_MBUS <= '0';
    Proc_Desire_L_MBUS <= '1';
    Add_In_OBM <= (others => 'Z');
    Data_WR_OBM <= (others => 'Z');
    RD_Req_OBM <= '0';
    WR_Req_OBM <= '0';
    Mem_req_Done <= '0';

  end if;
end process;

end Add_Select_arch;
```

```
==================================================================
 Event Bus Controller (State-Machine) <evt_fsm.vhdl>
==================================================================

 Project:            AYK-14 VHSIC Processor Module Hardware Emulator
 Component:          Event Bus Interface Controller
 Description:        State Machine that provides the interrogation of all polled Events
                     via the EBUS using control signals on the EMON Bus. Provides
                     capability to lock-out Class III interrupts via monitoring of SR1-
                     Bit3.Contains Timing loop that provides 9 clock cycles for each
                     state. This can be changed by calculating number of clock-cycles
                     required to permit a cycle time of 444 nsec.


 Author:             LT Bryan Fetter, USN
 Advisor:            Dr. Russ Duren
 Co-advisor:         Dr. Hersch Loomis
 Location:           Naval Postgraduate School

 Created:            25 October 2002
 Modified:           28 October 2002
 Simulated:
 Target:             XCV1000E FG1156
 Software:           Foundation 4.2i
 Notes:

Disclaimer:     NPS, makes no warranty for the use of this code or design. This code is
provided  "As Is". NPS, assumes no responsibility for  any errors, which may appear in
this code, nor does it make a commitment  to update the information contained herein.
NPS specifically disclaims any implied warranties of fitness for a particular purpose.
                Copyright (c) 2002 NPS
                All rights reserved.
==================================================================



library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use WORK.common.all;

entity EVT_FSM is
   port (
      EBUS: in STD_LOGIC_VECTOR (0 to 7);      -- Event Bus Input
      CLK: in STD_LOGIC;                       -- Clock
      RST: in STD_LOGIC;                       -- Reset
```

```
      SR1_BIT: in STD_LOGIC;                          -- Status Register 1 Bit 3
      EMON: out STD_LOGIC_VECTOR (0 to 7);  -- Event Monitor Bus
      E_VCTR: out STD_LOGIC_VECTOR (0 to 8)-- Event Vector (modified)
         );
end EVT_FSM;

architecture EVT_FSM_arch of EVT_FSM is

type evt_FSM_type is (Idle,Cls_Req, Grp_Req, Disc_Req);

constant Clock_Freq:  natural := 40_000_000;          --INPUT CLOCK FREQ in Hz
--***CHANGE THIS BASED ON OPERATING FREQ***
constant Design_Freq: natural := 40_000_000;          --Design Freq in Hz
constant Max_Cycles: natural := 9 * (Clock_Freq / Design_Freq);


signal curr_State, next_State: evt_FSM_type;

signal clk_count: unsigned(log2(Max_Cycles)-1 downto 0);
-- Used to count clock cycles
signal termCtrl: std_logic;
-- Used in counting process
signal Pri_Cls, Pri_Disc, Pri_Grp: std_logic_vector (2 downto 0);
--Used to generate Event Vector


begin
  -- Process to generate Next State

  nxt_St_Proc: process (curr_State, EBUS, SR1_BIT,clk_count)

  begin

  case curr_State is
          when Idle =>
            if (EBUS = "00000000") then          --No Events Active
              next_State <= Idle;
            else
             next_State <= Cls_Req;
            end if;
          when Cls_Req =>
            if (EBUS = "00000000") then          --No Events Active
              next_State <= Idle;
            --Non I/O Class 0
            elsif ((std_match(EBUS,"1-------"))
                    and clk_count = TO_UNSIGNED(Max_Cycles-1,clk_count'length)) then
```

```
                    next_State <= Disc_Req;
  --I/O Class 1
  elsif ((std_match(EBUS,"01------"))
            and clk_count = TO_UNSIGNED(Max_Cycles-1,clk_count'length)) then
                    next_State <= Grp_Req;
  --I/O Class 2
  elsif ((std_match(EBUS,"001-----"))
            and clk_count = TO_UNSIGNED(Max_Cycles-1,clk_count'length)) then
                    next_State <= Grp_Req;
  --Non I/O Class 3
  elsif ((std_match(EBUS,"0001----"))
            and clk_count = TO_UNSIGNED(Max_Cycles-1,clk_count'length)) then
                  next_State <= Disc_Req;
  --I/O Class 4
  elsif ((std_match(EBUS,"00001---"))
            and clk_count = TO_UNSIGNED(Max_Cycles-1,clk_count'length)) then
                    next_State <= Grp_Req;
  --Non I/O Class 5
  elsif ((std_match(EBUS,"000001--"))
            and clk_count = TO_UNSIGNED(Max_Cycles-1,clk_count'length)) then
    next_State <= Disc_Req;
  --Non I/O Class 6
  elsif ((std_match(EBUS,"0000001-"))
            and clk_count = TO_UNSIGNED(Max_Cycles-1,clk_count'length)) then
    next_State <= Disc_Req;
  --I/O Class 7
  elsif ((std_match(EBUS,"00000001"))
            and clk_count = TO_UNSIGNED(Max_Cycles-1,clk_count'length)
            and (SR1_BIT = '1')) then
    next_State <= Grp_Req;
  else
    next_State <= Cls_Req;
  end if;

when Grp_Req =>
-- Wait in this state for Max clocks
  if (clk_count = TO_UNSIGNED(Max_Cycles-1,clk_count'length)) then
    next_State <= Disc_Req;
  else
    next_State <= Grp_Req;
  end if;

when Disc_Req =>
-- Wait in this state for Max clocks
  if (clk_count = TO_UNSIGNED(Max_Cycles-1,clk_count'length)) then
    next_State <= Cls_req;
```

```
          else
            next_State <= Disc_Req;
          end if;

        when others =>
          null;

 end case;

 end process nxt_St_Proc;

--Current State Process - Clock triggered to make current state = next state

 curStProc: process (CLK, RST)
 begin
   if (RST = '1') then
     curr_State <= Idle;
   elsif (CLK'event and CLK = '1') then
     curr_State <= next_State;
   end if;
 end process curStProc;

-- Clock Counter - Provides 9 clock-cycles for each State when an event is active

 clock_counter: process (CLK, RST)
 begin
   case curr_State is
     when Idle =>
       clk_count <= TO_UNSIGNED(0,clk_count'length);
       termCtrl <= '1';
     when others =>

         if (CLK'event and CLK = '1') then

           if (termCtrl = '1') then
               clk_count <= TO_UNSIGNED(0,clk_count'length);
             else
               clk_count <= clk_count + 1;
             end if;

             if (clk_count = TO_UNSIGNED(Max_Cycles-1,clk_count'length)) then
               termCtrl <= '1';
             else
               termCtrl <= '0';
             end if;
```

```
        end if;
      end case;
  end process clock_counter;


--Output Conditioning Logic

 outConProc: process (curr_State, EBUS, Pri_Cls, Pri_Grp, Pri_Disc, SR1_BIT)
 begin
   case curr_State is
     when Idle =>
       EMON <= "01000000";
       --if (EBUS = "00000000") then
       --    Pri_Cls <= "000";
       --    Pri_Grp <= "000";
       --    Pri_Disc <= "000";
       --end if;

     when Cls_Req =>
       if (std_match(EBUS,"1-------")) then     --Non I/O Class 0
               Pri_Cls <= "000";
               Pri_Grp <= "000";
           elsif (std_match(EBUS,"01------")) then  --I/O Class 1
               Pri_Cls <= "001";
           elsif (std_match(EBUS,"001-----")) then  --I/O Class 2
               Pri_Cls <= "010";
           elsif (std_match(EBUS,"0001----")) then  --Non I/O Class 3
               Pri_Cls <= "011";
               Pri_Grp <= "000";
           elsif (std_match(EBUS,"00001---")) then  --I/O Class 4
               Pri_Cls <= "100";
           elsif (std_match(EBUS,"000001--")) then  --Non I/O Class 5
               Pri_Cls <= "101";
               Pri_Grp <= "000";
           elsif (std_match(EBUS,"0000001-")) then  --Non I/O Class 6
               Pri_Cls <= "110";
               Pri_Grp <= "000";
           elsif ((std_match(EBUS,"00000001"))
                     and (SR1_BIT = '1')) then        --I/O Class 7
               Pri_Cls <= "111";
           else
               Pri_Cls <= "000";
           end if;
       EMON <= "01000000";
     when Grp_Req =>
       if (std_match(EBUS,"1-------")) then     --Group 0/1
```

```
        Pri_Grp <= "000";
      elsif (std_match(EBUS,"01------")) then  --Group 2/3
        Pri_Grp <= "001";
      elsif (std_match(EBUS,"001-----")) then  --Group 4/5
        Pri_Grp <= "010";
      elsif (std_match(EBUS,"0001----")) then  --Group 6/7
        Pri_Grp <= "011";
      elsif (std_match(EBUS,"00001---")) then  --Group 8/9
        Pri_Grp <= "100";
      elsif (std_match(EBUS,"000001--")) then  --Group A/B
        Pri_Grp <= "101";
      elsif (std_match(EBUS,"0000001-")) then  --Group C/D
        Pri_Grp <= "110";
      elsif (std_match(EBUS,"00000001")) then  --Group E/F
        Pri_Grp <= "111";
      else
        Pri_Grp <= "000";
      end if;
      EMON <= "10" & Pri_Cls & "000";
   when Disc_Req =>
      if (std_match(EBUS,"1-------")) then     --Discrete 1 or Even 1
        Pri_Disc <= "000";
      elsif (std_match(EBUS,"01------")) then  --Discrete 2 or Even 2
        Pri_Disc <= "001";
      elsif (std_match(EBUS,"001-----")) then  --Discrete 3 or Even 3
        Pri_Disc <= "010";
      elsif (std_match(EBUS,"0001----")) then  --Discrete 4 or Even 4
        Pri_Disc <= "011";
      elsif (std_match(EBUS,"00001---")) then  --Discrete 5 or Odd 1
        Pri_Disc <= "100";
      elsif (std_match(EBUS,"000001--")) then  --Discrete 6 or Odd 2
        Pri_Disc <= "101";
      elsif (std_match(EBUS,"0000001-")) then  --Discrete 7 or Odd 3
        Pri_Disc <= "110";
      elsif (std_match(EBUS,"00000001")) then  --Discrete 8 or Odd 4
        Pri_Disc <= "111";
      else
        Pri_Disc <= "000";
      end if;
      EMON <= "11" & Pri_Cls & Pri_Grp;
   when others =>
      null;

  end case;

end process outConProc;
```

```
  E_VCTR <= Pri_Cls & Pri_Grp & Pri_Disc;

end EVT_FSM_arch;
```

```
=================================================================
-- SDRAM Controller <sdramcnt.vhdl>
=================================================================

-- Project:            AYK-14 VHSIC Processor Module Hardware Emulator
-- Component:          SDRAM Controller
-- Description:        State Machine that acts as the interface to the SDRAM and
                       provides all neccesary control and upkeep functions required for
                       SDRAM usage.


-- Author:             D. Van Den Bout
-- Modified for use in
-- this thesis by:     LT Bryan Fetter
-- Advisor:            Dr. Russ Duren
-- Co-advisor:         Dr. Hersch Loomis
-- Location:           Naval Postgraduate School

-- Modified:           27 November 2002
-- Simulated:          30 October 20020
-- Target:             XCV1000E FG1156
-- Software:           Foundation 4.2i
-- Notes:

Disclaimer:   NPS, makes no warranty for the use of this code or design. This code is
provided  "As Is". NPS, assumes no responsibility for  any errors, which may appear in
this code, nor does it make a commitment  to update the information contained herein.
NPS specifically disclaims any implied warranties of fitness for a particular purpose.
              Copyright (c) 2002 NPS
              All rights reserved.
=================================================================

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
--use unisim.vcomponents.all;
use WORK.common.all;
use WORK.xilinx.all;




package sdram is

component sdramCntl
        generic(
```

```
		FREQ:	natural := 40_000;-- operating frequency in KHz
		DATA_WIDTH: natural := 16;-- host & SDRAM data width
		NROWS: natural := 4096;	-- number of rows in SDRAM array
		NCOLS: natural := 512;		-- number of columns in SDRAM array
		HADDR_WIDTH: natural := 23;-- host-side address width
		SADDR_WIDTH: natural := 12 -- SDRAM-side address width
	);
	port(
		clkin:	in	std_logic;	-- master clock

		-- host side
		bufclk:	out std_logic;		-- buffered master clock
		clk0:		out std_logic;	-- host clock sync'ed to master clock
		clk2x:	out std_logic;		-- double-speed host clock
		lock:		out std_logic;	-- indicate when clock circuitry is
						-- locked to master clock
		rst:		in	std_logic;		-- reset
		rd:			in	std_logic;	-- read data
		wr:			in	std_logic;	-- write data
		done:		out	std_logic;		-- read/write op done
		hAddr:	in	unsigned(HADDR_WIDTH-1 downto 0);
		-- address from host
		hDIn:		in	unsigned(DATA_WIDTH-1 downto 0);
		-- data from host
		hDOut:		out	unsigned(DATA_WIDTH-1 downto 0);
		-- data to host
		sdramCntl_state: out std_logic_vector(3 downto 0);
		-- SDRAM side
		sclkfb: in	std_logic;	-- clock from SDRAM after PCB delays
		sclk:		out std_logic;	-- SDRAM clock sync'ed to master clock
		sclk_tst:	out std_logic;
		cke:		out	std_logic;-- clock-enable to SDRAM
		cs_n:		out	std_logic;-- chip-select to SDRAM
		ras_n:	out	std_logic;	-- command input to SDRAM
		cas_n:	out	std_logic;	-- command input to SDRAM
		we_n:		out	std_logic;-- command input to SDRAM
		ba:			out	unsigned(1 downto 0);
		-- SDRAM bank address bits
		sAddr: out	unsigned(SADDR_WIDTH-1 downto 0);
		-- SDRAM row/column address
		sData:	inout unsigned(DATA_WIDTH-1 downto 0);
		-- SDRAM in/out databus
		dqmh:		out	std_logic;		-- high databits I/O mask
		dqml:		out	std_logic		-- low databits I/O mask
	);
end component;
```

```
end package sdram;

library IEEE;--,unisim;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
--use unisim.vcomponents.all;
use WORK.common.all;
use WORK.xilinx.all;

entity sdramCntl is
	generic(
		FREQ: natural := 40_000;		-- operating frequency in KHz
		DATA_WIDTH: natural := 16;		-- host & SDRAM data width
		NROWS: natural := 4096;		-- number of rows in SDRAM array
		NCOLS: natural := 512;		-- number of columns in SDRAM
array
		HADDR_WIDTH: natural := 23;		-- host-side address width
		SADDR_WIDTH: natural := 12		-- SDRAM-side address width
	);
	port(
		clkin:	in	std_logic;	-- master clock

		-- host side
		bufclk: out std_logic;		-- buffered master clock
		clk0:		out std_logic;	-- host clock sync'ed to master clock
		clk2x:	out std_logic;		-- double-speed host clock
		lock:		out std_logic;	-- indicate when clock circuitry
						-- is locked to master clock
		rst:		in	std_logic;	-- reset
		rd:			in	std_logic;-- read data
		wr:			in	std_logic;-- write data
		done:		out	std_logic;	-- read/write op done
		hAddr: in	unsigned(HADDR_WIDTH-1 downto 0);
		-- address from host
		hDIn:		in	unsigned(DATA_WIDTH-1 downto 0);
		-- data from host
		hDOut:		out	unsigned(DATA_WIDTH-1 downto 0);
		-- data to host
		sdramCntl_state: out std_logic_vector(3 downto 0);

		-- SDRAM side
		sclkfb: in	std_logic;	-- clock from SDRAM after PCB delays
		sclk:		out std_logic;	-- SDRAM clock sync'ed to master clock
		sclk_tst:	out std_logic;
		cke:		out	std_logic;-- clock-enable to SDRAM
```

```
		cs_n:		out	std_logic;-- chip-select to SDRAM
		ras_n:	out	std_logic;	-- command input to SDRAM
		cas_n:	out	std_logic;	-- command input to SDRAM
		we_n:		out	std_logic;-- command input to SDRAM
		ba:			out	unsigned(1 downto 0);
		-- SDRAM bank address bits
		sAddr: out	unsigned(SADDR_WIDTH-1 downto 0);
		-- SDRAM row/column address
		sData:	inout unsigned(DATA_WIDTH-1 downto 0);
		-- SDRAM in/out databus
		dqmh:		out	std_logic;		-- high databits I/O mask
		dqml:		out	std_logic		-- low databits I/O mask
	);
end sdramCntl;



architecture arch of sdramCntl is

	-- constants
	constant ColCmdPos: natural := 10;
	-- position of command bit in SDRAM column address

	constant Tinit: natural := 100;-- min initialization interval (us)
	constant Tras:	natural := 44;	-- min interval between active
				to precharge commands (ns)
	constant Trc:		natural := 66;	-- min interval between active
					to active commands (ns)
	constant Trcd: natural := 20;		-- min interval between active
					and R/W commands (ns)
	constant Tref:	natural := 64_000_000;-- maximum refresh interval (ns)
	constant Trfc:	natural := 66;		-- duration of refresh operation (ns)
	constant Trp:		natural := 20;-- min precharge command duration (ns)
	constant Twr:		natural := 15;-- write recovery time (ns)
	constant Ccas: natural := 3;		-- CAS latency (cycles)
	constant Cmrd:		natural := 3;		-- mode register setup time (cycles)
	constant RfshCycles: natural := 8;	-- number of refresh cycles needed
					to init RAM

	constant ROW_LEN:			natural := log2(NROWS);
	-- number of row address bits
	constant COL_LEN:			natural := log2(NCOLS);
	-- number of column address bits
	constant NORM:				natural := 1_000_000;
	-- normalize ns * KHz
	constant INIT_CYCLES:	natural := 1 + ((Tinit * FREQ) / 1000);
```

```
-- SDRMA power-on initialization interval
constant RAS_CYCLES:       natural := 1 + ((Tras * FREQ) / NORM);
-- active-to-precharge interval
constant RC_CYCLES:                natural := 1 + ((Trc * FREQ) / NORM);
-- active-to-active interval
constant RCD_CYCLES:       natural := 1 + ((Trcd * FREQ) / NORM);
-- active-to-R/W interval
constant REF_CYCLES:       natural := 1 + (((Tref/NROWS) * FREQ) / NORM);
-- interval between row refreshes
constant RFC_CYCLES:       natural := 1 + ((Trfc * FREQ) / NORM);
-- refresh operation interval
constant RP_CYCLES:                natural := 1 + ((Trp * FREQ) / NORM);
-- precharge operation interval
constant WR_CYCLES:                natural := 1 + ((Twr * FREQ) / NORM);
-- write recovery time

-- states of the SDRAM controller state machine
type cntlState is (
        INITWAIT,   -- initialization –
                         waiting for power-on initialization to complete
        INITPCHG,   -- initialization - doing precharge of banks
        INITSETMODE,-- initialization - set SDRAM mode
        INITRFSH,   -- initialization - do refreshes
        REFRESH,    -- refresh a row of the SDRAM
        RW,         -- wait for read/write operations to SDRAM
        RDDONE,         -- indicate that the SDRAM read is done
        WRDONE,         -- indicate that the SDRAM write is done
        ACTIVATE    -- open a row of the SDRAM for reading/writing
);
signal state_r, state_next: cntlState;   -- state register and next state

constant AUTO_PCHG_ON: std_logic := '1';
-- set sAddr(10) to this value to auto-precharge the bank
constant AUTO_PCHG_OFF:          std_logic := '0';
-- set sAddr(10) to this value to disable auto-precharge
constant ALL_BANKS:                std_logic := '1';
-- set sAddr(10) to this value to select all banks
constant ACTIVE_BANK:     std_logic := '0';
-- set sAddr(10) to this value to select only the active bank
signal bank: unsigned(ba'range);
signal row: unsigned(ROW_LEN - 1 downto 0);
signal col: unsigned(COL_LEN - 1 downto 0);
signal col_tmp: unsigned(sAddr'high-1 downto sAddr'low);
signal changeRow: std_logic;
signal dirOut: std_logic; -- high when driving data to SDRAM
```

```
-- registers
signal activeBank_r, activeBank_next: unsigned(bank'range);
-- currently active SDRAM bank
signal activeRow_r, activeRow_next: unsigned(row'range);
-- currently active SDRAM row
signal inactiveFlag_r, inactiveFlag_next: std_logic;
-- 1 when all SDRAM rows are inactive
signal doRfshFlag_r, doRfshFlag_next: std_logic;
-- 1 when a row refresh operation is required
signal wrFlag_r, wrFlag_next: std_logic;
-- 1 when writing data to SDRAM
signal rdFlag_r, rdFlag_next: std_logic;
-- 1 when reading data from SDRAM
signal rfshCntr_r, rfshCntr_next: unsigned(log2(RfshCycles+1)-1 downto 0);
-- counts initialization refreshes

-- timer registers that count down times for various SDRAM operations
signal timer_r, timer_next: unsigned(log2(INIT_CYCLES+1)-1 downto 0);
-- current SDRAM op time
signal rasTimer_r, rasTimer_next: unsigned(log2(RAS_CYCLES+1)-1
                                        downto 0);
-- active-to-precharge time
signal wrTimer_r, wrTimer_next: unsigned(log2(WR_CYCLES+1)-1 downto 0);
-- write-to-precharge time
signal refTimer_r, refTimer_next: unsigned(log2(REF_CYCLES+1)-1 downto 0);
-- time between row refreshes

-- SDRAM commands
subtype sdramCmd is unsigned(5 downto 0);
-- cmd = (cs_n,ras_n,cas_n,we_n,dqmh,dqml)
constant NOP_CMD:          sdramCmd := "011100";
constant ACTIVE_CMD:       sdramCmd := "001100";
constant READ_CMD:         sdramCmd := "010100";
constant WRITE_CMD:        sdramCmd := "010000";
constant PCHG_CMD:         sdramCmd := "001011";
constant MODE_CMD:         sdramCmd := "000011";
constant RFSH_CMD:         sdramCmd := "000111";
signal cmd: sdramCmd;

-- SDRAM mode register
subtype sdramMode is unsigned(11 downto 0);
constant MODE: sdramMode := "00" & "0" & "00" & "011" & "0" & "000";

-- clock DLL signals
signal logic0: std_logic;
-- signals for internal logic clock DLL
```

```
	signal  bufclkin, dllint_clk0, dllint_clk2x, bufdllint_clk0,
		bufdllint_clk2x, lockint: std_logic;
	-- signals for external logic clock DLL
	signal bufdllext_clk0, dllext_clk0, lockext: std_logic;
	signal clk: std_logic;   -- clock for SDRAM controller logic

begin

	logic0 <= '0';

	-- master clock must come from a dedicated clock pin
	clkpad: IBUFG port map (I=>clkin, O=>bufclkin);
	bufclk <= bufclkin;

	-- generate an internal clock sync'ed to the master clock
	dllint: CLKDLL port map(
		CLKIN=>bufclkin, CLKFB=>bufdllint_clk0, CLK0=>dllint_clk0,
		RST=>logic0, CLK90=>open, CLK180=>open, CLK270=>open,
		CLK2X=>dllint_clk2x, CLKDV=>open, LOCKED=>lockint
	);
	-- sync'ed single and double-speed clocks for use by internal logic
	clkg: BUFG port map (I=>dllint_clk0, O=>bufdllint_clk0);
	clkg2x: BUFG port map(I=>dllint_clk2x, O=>bufdllint_clk2x);
	clk <= bufdllint_clk0;		-- SDRAM controller logic clock
	clk0 <= bufdllint_clk0;			-- clock to other FPGA logic
	clk2x <= bufdllint_clk2x;	-- doubled clock to other FPGA logic;
	lock <= lockint and lockext;	-- indicate lock status of the DLLs

	-- generate an external SDRAM clock sync'ed to the master clock
--	clkfbpad : IBUFG  port map (I=>sclkfb, O=>bufsclkfb);  -- SDRAM clock with
PCB delays
--	dllext: CLKDLL port map(
--		CLKIN=>bufclkin, CLKFB=>bufsclkfb, CLK0=>dllext_clk0,
	clkfbpad : BUFG  port map (I=>dllext_clk0, O=>bufdllext_clk0);
	-- SDRAM clock with PCB delays
	dllext: CLKDLL port map(
		CLKIN=>bufclkin, CLKFB=>bufdllext_clk0, CLK0=>dllext_clk0,
		RST=>logic0, CLK90=>open, CLK180=>open, CLK270=>open,
		CLK2X=>open, CLKDV=>open, LOCKED=>lockext
	);

	-- output the sync'ed SDRAM clock to the SDRAM
	clkextpad: OBUF port map (I=>dllext_clk0, O=>sclk);
	clkextpad_2: OBUF port map (I=>bufdllext_clk0, O=>sclk_tst);
```

```
hDOut <= sData(hDOut'range);        -- connect SDRAM data bus to host data bus
sData <= hDIn(sData'range) when dirOut='1' else (others=>'Z');     \
-- connect host data bus to SDRAM data bus

combinatorial: process(rd,wr,hAddr,hDIn,state_r,bank,row,col,changeRow,
        activeBank_r,activeRow_r,doRfshFlag_r,rdFlag_r,wrFlag_r,

rfshCntr_r,timer_r,rasTimer_r,wrTimer_r,refTimer_r,cmd,col_tmp,inactiveFlag_r
)
begin
        -- attach bits in command to SDRAM control signals
        (cs_n,ras_n,cas_n,we_n,dqmh,dqml) <= cmd;

        -- get bank, row, column from host address
        bank <= hAddr(bank'length + ROW_LEN + COL_LEN - 1
                downto ROW_LEN + COL_LEN);
        row <= hAddr(ROW_LEN + COL_LEN - 1 downto COL_LEN);
        col <= hAddr(COL_LEN - 1 downto 0);
        -- extend column (if needed) until it is as large
           as the (SDRAM address bus - 1)
        col_tmp <= (others=>'0');       -- set it to all zeroes
        col_tmp(col'range) <= col;      -- write column into the lower bits

        -- default operations
        cke <= YES;   -- enable SDRAM clock input
        cmd <= NOP_CMD;   -- set SDRAM command to no-operation
        done <= NO;   -- pending SDRAM operation is not done
        ba <= bank;   -- set SDRAM bank address bits
        -- set SDRAM address to column with interspersed command bit
        sAddr(ColCmdPos-1 downto 0) <= col_tmp(ColCmdPos-1 downto 0);
        sAddr(sAddr'high downto ColCmdPos+1) <=
                col_tmp(col_tmp'high downto ColCmdPos);
        sAddr(ColCmdPos) <= AUTO_PCHG_OFF;
        -- set command bit to disable auto-precharge
        dirOut <= NO;

        -- default register updates
        state_next <= state_r;
        inactiveFlag_next <= inactiveFlag_r;
        activeBank_next <= activeBank_r;
        activeRow_next <= activeRow_r;
        doRfshFlag_next <= doRfshFlag_r;
        rdFlag_next <= rdFlag_r;
        wrFlag_next <= wrFlag_r;
        rfshCntr_next <= rfshCntr_r;
```

```
-- update timers
if timer_r /= TO_UNSIGNED(0,timer_r'length) then
        timer_next <= timer_r - 1;
else
        timer_next <= timer_r;
end if;

if rasTimer_r /= TO_UNSIGNED(0,rasTimer_r'length) then
        rasTimer_next <= rasTimer_r - 1;
else
        rasTimer_next <= rasTimer_r;
end if;

if wrTimer_r /= TO_UNSIGNED(0,wrTimer_r'length) then
        wrTimer_next <= wrTimer_r - 1;
else
        wrTimer_next <= wrTimer_r;
end if;

if refTimer_r /= TO_UNSIGNED(0,refTimer_r'length) then
        refTimer_next <= refTimer_r - 1;
else
-- on timeout, reload the timer with the interval between row refreshes
-- and set the flag that indicates a refresh operation is needed.
refTimer_next<=
        TO_UNSIGNED(REF_CYCLES,refTimer_next'length);
        doRfshFlag_next <= YES;
end if;

-- determine if another row or bank in the SDRAM is being addressed
if row /= activeRow_r or bank /= activeBank_r
  or inactiveFlag_r = YES then
        changeRow <= YES;
else
        changeRow <= NO;
end if;

-- ***** compute next state and outputs *****

-- SDRAM initialization

-- don't do anything if the previous operation has not completed yet.
-- Place this before anything else so operations in the previous state
-- complete before any operations in the new state are executed.
if timer_r /= TO_UNSIGNED(0,timer_r'length) then
        sdramCntl_state <= "0000";
```

```
elsif state_r = INITWAIT then
        -- initiate wait for SDRAM power-on initialization
        timer_next
                <= TO_UNSIGNED(INIT_CYCLES,timer_next'length);
        -- set timer for init interval
        state_next <= INITPCHG;
        -- precharge SDRAM after power-on initialization
        sdramCntl_state <= "0001";
elsif state_r = INITPCHG then
        cmd <= PCHG_CMD;        -- initiate precharge of the SDRAM
        sAddr(ColCmdPos) <= ALL_BANKS;      -- precharge all banks
        timer_next <= TO_UNSIGNED(RP_CYCLES,timer_next'length);
        -- set timer for this operation
        -- now setup the counter for the number of refresh ops
        -- needed during initialization
        rfshCntr_next <=
                TO_UNSIGNED(RfshCycles,rfshCntr_next'length);
        state_next <= INITRFSH;
        -- perform refresh ops after setting the mode
        sdramCntl_state <= "0010";
elsif state_r = INITRFSH then
        -- refresh the SDRAM a number of times during initialization
        if rfshCntr_r /= TO_UNSIGNED(0,rfshCntr_r'length) then
                -- do a refresh operation if the counter is not zero yet
                cmd <= RFSH_CMD;-- refresh command goes to SDRAM
                timer_next <=
                        TO_UNSIGNED(RFC_CYCLES,timer_next'length);
                -- refresh operation interval
                rfshCntr_next <= rfshCntr_r - 1;
                -- decrement refresh operation counter
                state_next <= INITRFSH;
                -- return to this state while counter is non-zero
        else
                -- refresh op counter reaches zero,
                -- so set the operating mode of the SDRAM
                state_next <= INITSETMODE;
        end if;
        sdramCntl_state <= "0100";
elsif state_r = INITSETMODE then
        -- set the mode register in the SDRAM
        cmd <= MODE_CMD;
        -- initiate loading of mode register in the SDRAM
        sAddr <= MODE;
        -- output mode register bits onto the SDRAM address bits
        timer_next <= TO_UNSIGNED(Cmrd,timer_next'length);
```

```
            -- set timer for this operation
            state_next <= RW;
            -- process read/write operations after initialization is done
            sdramCntl_state <= "0011";

-- refresh a row of the SDRAM when the refresh timer hits zero and
   sets the flag
-- and the SDRAM is no longer being read/written.
-- Place this before the RW state so the host can't block refreshes by doing
-- continuous read/write operations.
elsif doRfshFlag_r = YES and wrFlag_r = NO and rdFlag_r = NO then
            if rasTimer_r = TO_UNSIGNED(0,rasTimer_r'length)
               and wrTimer_r = TO_UNSIGNED(0,wrTimer_r'length) then
                        doRfshFlag_next <= NO;
                        -- reset the flag that initiates a refresh operation
                        cmd <= PCHG_CMD;
                        -- initiate precharge of the SDRAM
                        sAddr(ColCmdPos) <= ALL_BANKS;
                        -- precharge all banks
                        timer_next <=
                           TO_UNSIGNED(RP_CYCLES,timer_next'length);
                        -- set timer for this operation
                        inactiveFlag_next <= YES;
                        -- all rows are inactive after a precharge operation
                        state_next <= REFRESH;
                        -- refresh the SDRAM after the precharge
            end if;
            sdramCntl_state <= "0101";
elsif state_r = REFRESH then
            cmd <= RFSH_CMD;-- refresh command goes to SDRAM
            timer_next <=
                  TO_UNSIGNED(RFC_CYCLES,timer_next'length);
            -- refresh operation interval
            -- after refresh is done, resume writing or reading the SDRAM
                if in progress
            state_next <= RW;
            sdramCntl_state <= "0110";

-- do nothing but wait for read or write operations
elsif state_r = RW then
            if rd = YES then
                        -- the host has initiated a read operation
                        rdFlag_next <= YES;
                        -- set flag to indicate a read operation is in progress
                        -- if a different row or bank is being read,
                        -- then precharge the SDRAM and activate the new row
```

```
                    if changeRow = YES then
                    -- wait for any row activations or writes to
                    -- finish before doing a precharge
                            if rasTimer_r
                             = TO_UNSIGNED(0,rasTimer_r'length)
                            and wrTimer_r
                            = TO_UNSIGNED(0,wrTimer_r'length) then
                                    cmd <= PCHG_CMD;
                                    -- initiate precharge of the SDRAM
                                    sAddr(ColCmdPos) <= ALL_BANKS;
                                    -- precharge all banks
                                    timer_next <=
                                    TO_UNSIGNED(RP_CYCLES,
                                            timer_next'length);
                                    -- set timer for this operation
                                    inactiveFlag_next <= YES;
                                    -- all rows are inactive after a
                                       precharge operation
                                    state_next <= ACTIVATE;
                                    -- activate the new row after the
                                       precharge is done
                            end if;
                    -- read from the currently active row
                    else
                            cmd <= READ_CMD;
                            -- initiate a read of the SDRAM
                            timer_next <=
                                TO_UNSIGNED(Ccas,timer_next'length);
                            -- setup timer for read access
                            state_next <= RDDONE;
                            -- read the data from SDRAM after the access time
                    end if;
                    sdramCntl_state <= "0111";
            elsif wr = YES then
                    -- the host has initiated a write operation
                    -- if a different row or bank is being written,
                    -- then precharge the SDRAM and activate the new row
                    if changeRow = YES then
                            wrFlag_next <= YES;
                            -- set flag to indicate a write operation is in progress
                            -- wait for any row activations or writes to finish
                            -- before doing a precharge
                            if rasTimer_r =
                                TO_UNSIGNED(0,rasTimer_r'length)
                              and wrTimer_r =
                                TO_UNSIGNED(0,wrTimer_r'length) then
```

```
                    cmd <= PCHG_CMD;
                    -- initiate precharge of the SDRAM
                    sAddr(ColCmdPos) <= ALL_BANKS;
                    -- precharge all banks
                    timer_next <=
                    TO_UNSIGNED(RP_CYCLES,
                                  timer_next'length);
                    -- set timer for this operation
                    inactiveFlag_next <= YES;
                    -- all rows are inactive after a
                       precharge operation
                    state_next <= ACTIVATE;
                    -- activate the new row after
                       the precharge is done
                end if;
            -- write to the currently active row
            else
                cmd <= WRITE_CMD;
                -- initiate the write operation
                dirOut <= YES;
                -- set timer so precharge doesn't occur
                -- too soon after write operation
                wrTimer_next <=
                TO_UNSIGNED(WR_CYCLES,
                              wrTimer_next'length);
                state_next <= WRDONE;
                -- go back and wait for another read/write operation
            end if;
            sdramCntl_state <= "1000";
        else
            null;    -- no read or write operation, so do nothing
            sdramCntl_state <= "1001";
        end if;

    -- enter this state when the data read from the SDRAM is available
    elsif state_r = RDDONE then
        rdFlag_next <= NO;-- set flag to indicate the read operation is over
        done <= YES;   -- tell the host that the data is ready
        state_next <= RW; -- go back and do another read/write operation
        sdramCntl_state <= "1010";

    -- enter this state when the data is written to the SDRAM
    elsif state_r = WRDONE then
        dirOut <= YES;
        wrFlag_next <= NO;
        -- set flag to indicate the write operation is over
```

```
			done <= YES;   -- tell the host that the data is ready
			state_next <= RW; -- go back and do another read/write operation
			sdramCntl_state <= "1011";

		-- activate a row of the SDRAM
		elsif state_r = ACTIVATE then
			cmd <= ACTIVE_CMD;
			-- initiate the SDRAM activation operation
			sAddr <= (others=>'0');
			-- output the address for the row that will be activated
			sAddr(row'range) <= row;
			activeBank_next <= bank;-- remember the active SDRAM row
			activeRow_next <= row;
			-- remember the active SDRAM bank
			inactiveFlag_next <= NO;-- the SDRAM is no longer inactive
			rasTimer_next <=
				TO_UNSIGNED(RCD_CYCLES,rasTimer_next'length);
			timer_next <=
				TO_UNSIGNED(RCD_CYCLES,timer_next'length);
			state_next <= RW;
			-- go back and do the read/write operation that
			   caused this activation
			sdramCntl_state <= "1100";

		-- no operation
		else
			null;
			sdramCntl_state <= "1101";

		end if;

	end process combinatorial;


	-- update registers on the rising clock edge
	update: process(clk)
	begin
	  if clk'event and clk='1' then
		if rst = YES then
			state_r			<= INITWAIT;
			activeBank_r		<= (others=>'0');
			activeRow_r		<= (others=>'0');
			inactiveFlag_r		<= YES;
			doRfshFlag_r		<= NO;
			rdFlag_r		<= NO;
			wrFlag_r		<= NO;
```

```
                rfshCntr_r          <= TO_UNSIGNED(0,rfshCntr_r'length);
                timer_r             <= TO_UNSIGNED(0,timer_r'length);
                refTimer_r     <=
                        TO_UNSIGNED(REF_CYCLES,refTimer_r'length);
                rasTimer_r          <= TO_UNSIGNED(0,rasTimer_r'length);
                wrTimer_r           <= TO_UNSIGNED(0,wrTimer_r'length);
            else
                state_r             <= state_next;
                activeBank_r        <= activeBank_next;
                activeRow_r         <= activeRow_next;
                inactiveFlag_r      <= inactiveFlag_next;
                doRfshFlag_r        <= doRfshFlag_next;
                rdFlag_r            <= rdFlag_next;
                wrFlag_r            <= wrFlag_next;
                rfshCntr_r          <= rfshCntr_next;
                timer_r             <= timer_next;
                refTimer_r          <= refTimer_next;
                rasTimer_r          <= rasTimer_next;
                wrTimer_r           <= wrTimer_next;
            end if;
        end if;
    end process update;

end arch;
```

```
=====================================================================
 xs_package <xs_pckg.vhd>
=====================================================================

 Project:          AYK-14 VHSIC Processor Module Hardware Emulator
 Component:        Commom Component Declaration
 Description:      Declaration of simple components needed in other
                   components.

 Author:           D. Van den Bout
 Adapted by:       LT Bryan Fetter
 Advisor:          Dr. Russ Duren
 Co-advisor:       Dr. Hersch Loomis
 Location:         Naval Postgraduate School

 Created:          1 September 2002
 Modified:         7 November 2002
 Simulated:
 Target:           XCV1000E FG1156
 Software:         Foundation 4.2i
 Notes:

Disclaimer:   NPS, makes no warranty for the use of this code or design. This code is
provided  "As Is". NPS, assumes no responsibility for  any errors, which may appear in
this code, nor does it make a commitment  to update the information contained herein.
NPS specifically disclaims any implied warranties of fitness for a particular purpose.
              Copyright (c) 2002 NPS
              All rights reserved.
=====================================================================

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

package common is

      constant YES: std_logic := '1';
      constant NO:  std_logic := '0';
      constant HI:  std_logic := '1';
      constant LO:  std_logic := '0';
      function log2(v: in natural) return natural;

end package common;
```

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;


package body common is

function log2(v: in natural) return natural is
	variable n: natural;
	variable logn: natural;
begin
	n := 1;
	for i in 0 to 128 loop
		logn := i;
		exit when (n>=v);
		n := n * 2;
	end loop;
	return logn;
end function log2;

end package body common;



library IEEE;--,VIRTEX;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
--use VIRTEX.components.all;

package xilinx is

component IBUFG
	port(
		O:	out	std_ulogic;
		I:	in	std_ulogic
	);
end component;

component CLKDLL
	port(
		CLKIN:		in  std_ulogic := '0';
		CLKFB:		in  std_ulogic := '0';
		RST:		in  std_ulogic := '0';
		CLK0:		out std_ulogic := '0';
		CLK90:		out std_ulogic := '0';
		CLK180:		out std_ulogic := '0';
```

```
		CLK270:		out std_ulogic := '0';
		CLK2X:		out std_ulogic := '0';
		CLKDV:		out std_ulogic := '0';
		LOCKED:		out std_ulogic := '0'
	);
end component;

component BUFG
	port(
		O:	out	std_ulogic;
		I:	in	std_ulogic
	);
end component;

component OBUF
	port(
		O:	out	std_ulogic;
		I:	in	std_ulogic
	);
end component;

end package xilinx;
```

==
Odd Parity Generator <oddparity.vhd.vhd>
==

Project:	AYK-14 VHSIC Processor Module Hardware Emulator
Component:	Odd Parity Generator
Description:	Odd parity generator adapted from a design in "Essential VHDL" by Sundar Rajan. Generates sets of XORs and connects them to the bits of the incoming Byte to generate odd parity
Author:	Sundar Rajan
Adapted by:	LT Bryan Fetter, USN
Advisor:	Dr. Russ Duren
Co-advisor:	Dr. Hersch Loomis
Location:	Naval Postgraduate School
Created:	25 October 2002
Modified:	24 November 2002
Simulated:	
Target:	XCV1000E FG1156
Software:	Foundation 4.2i
Notes:	

==

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

package oddParity is

component oddParityGen
   generic( width : integer := 8);
   port (
      data: in UNSIGNED (width - 1 downto 0);
      parity: out STD_LOGIC
   );
end component;

end package oddParity;
```

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity oddParityGen is
    generic( width : integer := 8);
    port (
        data: in UNSIGNED (width - 1 downto 0);
        parity: out STD_LOGIC
    );
end oddParityGen;

architecture oddParityGen_arch of oddParityGen is
begin

  process (data)
    variable loopXor: std_logic;
  begin
    loopXor := '0';

    for i in 0 to width -1 loop
      loopXor := loopXor xor data(i);
    end loop;

    parity <= loopXor;

  end process;

end oddParityGen_arch;
```

==

MBUS Desire / Grant Arbitrator <grant_logic.vhd>

==

Project:	AYK-14 VHSIC Processor Module Hardware Emulator
Component:	MBUS Grant Arbitrator
Description:	State machine that provides rotating priority logic to determine the next user of the MBUS. The component analyzes the MBUS Request signals from the 3 MBUS users and provides MBUS Grant signals to the appropriate user. The priority is a rotating type that ensures that each user has equal access to the bus based upon the previous user.

Author:	LT Bryan Fetter, USN
Advisor:	Dr. Russ Duren
Co-advisor:	Dr. Hersch Loomis
Location:	Naval Postgraduate School

Created:	25 October 2002
Modified:	7 November 2002
Simulated:	
Target:	XCV1000E FG1156
Software:	Foundation 4.2i
Notes:	

==

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
--use IEEE.std_logic_unsigned.all;
--use IEEE.std_logic_arith.all;


package Grant is

component Grant_Logic
   port (
      M_Desire_Ext: in UNSIGNED (1 downto 0);
      M_Desire_Proc: in STD_LOGIC;
```

```
        M_Grant_Ext: out UNSIGNED (1 downto 0);
           M_Grant_Proc: out STD_LOGIC;
        Clk: in STD_LOGIC;
        Rst: in STD_LOGIC
    );
end component;

end package Grant;

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
--use IEEE.std_logic_unsigned.all;
--use IEEE.std_logic_arith.all;

entity Grant_Logic is
    port (
        M_Desire_Ext: in UNSIGNED (1 downto 0);
        M_Desire_Proc: in STD_LOGIC;
        M_Grant_Ext: out UNSIGNED (1 downto 0);
           M_Grant_Proc: out STD_LOGIC;
        Clk: in STD_LOGIC;
        Rst: in STD_LOGIC
    );
end Grant_Logic;

architecture Grant_Logic_arch of Grant_Logic is

type FSM_type is (Idle,Grant);
signal Curr_State, Next_State : FSM_Type;
signal User : UNSIGNED (1 downto 0);
signal Pri_0,Pri_1,Pri_2 : UNSIGNED (1 downto 0);

signal M_Desire_Int : UNSIGNED (2 downto 0);
signal M_Grant_Int : UNSIGNED (2 downto 0);


begin

M_Desire_Int(1) <= M_Desire_Ext(1);
M_Desire_Int(0) <= M_Desire_Ext(0);
M_Desire_Int(2) <= M_Desire_Proc;

M_Grant_Ext(1) <= M_Grant_Int(1);
M_Grant_Ext(0) <= M_Grant_Int(0);
M_Grant_Proc <= M_Grant_Int(2);
```

```
nxtStProc: process(Curr_State,Next_State, M_Desire_Int, User)

  begin

      case Curr_State is


        when Idle =>

          if M_Desire_Int /= "111" then
            Next_State <= Grant;
          else
            Next_State <= Idle;
          end if;

        when Grant =>
          if (M_Desire_Int(to_integer(User)) = '0') then
                Next_State <= Grant;
              else
                Next_State <= Idle;
              end if;

            when others =>
              null;

       end case;
     end process nxtStProc;

--Process to register current state

  curStProc: process (Clk, Rst)
  begin
    if (Rst = '0') then
       Curr_State <= Idle;
    elsif (Clk'event and Clk ='1') then
       Curr_State <= Next_State;
    end if;
  end process curStProc;

--Process to generate outputs

  outConProc: process(Curr_State,M_Desire_Int,Pri_0,Pri_1,Pri_2,User)

  begin
```

```
case Curr_State is

when Idle =>
   M_Grant_Int <= "000";

   --to handle Reset
   if (Pri_0 = Pri_1) then
     if ((M_Desire_Int(0)) = '0' )then
       User <= "00";
     elsif ((M_Desire_Int(1)) = '0' )then
       User <= "01";
     elsif ((M_Desire_Int(2)) = '0' )then
       User <= "10";
     end if;
   elsif (M_Desire_Int(to_integer(Pri_0)) = '0' )then
     User <= Pri_0;
   elsif (M_Desire_Int(to_integer(Pri_1)) = '0' )then
     User <= Pri_1;
   elsif (M_Desire_Int(to_integer(Pri_2)) = '0' )then
     User <= Pri_2;
   end if;


 when Grant =>
   M_Grant_Int(to_integer(User)) <= '1';

   if User = "00" then
     Pri_0 <= "01";
     Pri_1 <= "10";
     Pri_2 <= "00";
   elsif User = "01" then
     Pri_0 <= "10";
     Pri_1 <= "00";
     Pri_2 <= "01";
   elsif User = "10" then
     Pri_0 <= "00";
     Pri_1 <= "01";
     Pri_2 <= "10";
   else
     Pri_0 <= "00";
     Pri_1 <= "01";
     Pri_2 <= "10";
   end if;

 when others =>
```

```
      null;
    end case;
  end process outConProc;
end Grant_Logic_arch;
```

```
==================================================================
```

MBUS Controller <mbus_controller.vhd>

```
==================================================================
```

Project:	AYK-14 VHSIC Processor Module Hardware Emulator
Component:	MBUS Controller
Description:	State Machine that controls the MBUS interface. It determines the user of the bus via the Grant_Logic component and generates the appropriate control signals for operation of the Bus for reads and writes both to OBM by an external user as well as reads and writes to external memory by the Processor. It also generates and validates the appropriate parity signals.

Author:	LT Bryan Fetter, USN
Advisor:	Dr. Russ Duren
Co-advisor:	Dr. Hersch Loomis
Location:	Naval Postgraduate School
Created:	25 October 2002
Modified:	23 November 2002
Simulated:	27 November 2002
Target:	XCV1000E FG1156
Software:	Foundation 4.2i
Notes:	


```
==================================================================
```

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use WORK.Grant.all;
use WORK.common.all;
use WORK.oddParity.all;
--use IEEE.std_logic_arith.all;

package MBUS_CTRL is

component MBUS_Controller
   generic(
            FREQ: natural := 40_000  -- operating frequency in KHz
```

```
);
port (
   Clk: in std_logic;
   Rst: in std_logic;
   -- Signals from Processor
   P_Data_WR:      in unsigned(31 downto 0);
   P_Data_RD:      out unsigned(31 downto 0);
   P_Addr:  in unsigned(22 downto 0);
   P_RD_Req:       in std_logic;
   P_WR_Req:       in std_logic;
   P_Desire_L:     in std_logic;
   P_Mem_Done:     out STD_LOGIC;
   P_Grant_Out:    out std_logic;  --Grant signal to Processor

   -- Signals from Memory Arbitrator
   Mem_Addr:       out unsigned(22 downto 0);
   Mem_Data_WR:out unsigned(31 downto 0);
   Mem_Data_RD: in unsigned(31 downto 0);
   Mem_WR_Req: out std_logic;
   Mem_RD_Req:  out std_logic;
   Mem_Done:       in std_logic;
   -- Signals on/off Adapter
   M_BUS:          inout unsigned(22 downto 0);
   --M_GRANT_IN_L:       in std_logic;    Used only when used as Slave
   M_DESIRE_IN_L:      in unsigned(1 downto 0);
   M_GRANT_OUT:        out unsigned(1 downto 0);
   --M_DESIRE_OUT_L: out std_logic;--Used only when VPM used as Slave
   M_REQUEST_L:        inout std_logic;
   M_ACKNOWLEDGE_L:in std_logic;
   M_RESUME_L: inout std_logic;
   S_BUSY_L:       out std_logic;
   M_BUSY_L:       inout std_logic;
   BUS_ERROR_L:        inout std_logic;
      --Parity Bits
   LSB_PARITY:   inout std_logic;
   MSB_PARITY:  inout std_logic;
   ADRS_PARITY:        inout std_logic;
   CMD_PARITY: inout std_logic;
      --Control Bits
   MSB_WRITE_L:        inout std_logic;
   LSB_WRITE_L:inout std_logic;
   THREE_TWO_DATA:  inout std_logic;
   IPL_WRITE:      inout std_logic;

   --Signals used for Testing Only
   Timer_Out:      out unsigned(log2(9+1)-1 downto 0);
```

```
        Timer_next_Out: out unsigned(log2(9+1)-1 downto 0);
        M_ACKNOWLEDGE_L_test_Out:      out std_logic

    );
end component;

end MBUS_Ctrl;

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use WORK.Grant.all;
use WORK.Common.all;
use WORK.oddParity.all;--use IEEE.std_logic_arith.all;

entity MBUS_Controller is
    generic(
        FREQ:    natural := 40_000 -- operating frequency in KHz
    );
    port (
        Clk: in std_logic;
        Rst: in std_logic;
        -- Signals from Processor
        P_Data_WR:      in unsigned(31 downto 0);
        P_Data_RD:      out unsigned(31 downto 0);
        P_Addr:  in unsigned(22 downto 0);
        P_RD_Req:       in std_logic;
        P_WR_Req:       in std_logic;
        P_Desire_L:     in std_logic;
        P_Mem_Done:     out STD_LOGIC;
        P_Grant_Out:    out std_logic;  --Grant signal to Processor

        -- Signals from Memory Arbitrator
        Mem_Addr:       out unsigned(22 downto 0);
        Mem_Data_WR:    out unsigned(31 downto 0);
        Mem_Data_RD:    in unsigned(31 downto 0);
        Mem_WR_Req:     out std_logic;
        Mem_RD_Req:     out std_logic;
        Mem_Done:       in std_logic;
        -- Signals on/off Adapter
        M_BUS:          inout unsigned(22 downto 0);
--      M_GRANT_IN_L:        in std_logic;    --Used only when VPM used as Slave
        M_DESIRE_IN_L:       in unsigned(1 downto 0);
        M_GRANT_OUT:         out unsigned(1 downto 0);
--      M_DESIRE_OUT_L: out std_logic;--Used only when VPM used as Slave
        M_REQUEST_L:         inout std_logic;
```

```
        M_ACKNOWLEDGE_L:in std_logic;
        M_RESUME_L: inout std_logic;
        S_BUSY_L:       out std_logic;
        M_BUSY_L:       inout std_logic;
        BUS_ERROR_L:            inout std_logic;
            --Parity Bits
        LSB_PARITY:     inout std_logic;--Odd Parity for Bits MBUS(0:7)
        MSB_PARITY:     inout std_logic;--Odd Parity for Bits MBUS(8:15)
        ADRS_PARITY:            inout std_logic;--Odd Parity for Bits MBUS(16:22)
        CMD_PARITY:     inout std_logic;--Odd Parity for
                                --MSB_Write/LSB_Write/32_Bit_Data/IPL_Write
            --Control Bits
        MSB_WRITE_L:            inout std_logic;
        LSB_WRITE_L: inout std_logic;
        THREE_TWO_DATA: inout std_logic;
        IPL_WRITE:      inout std_logic;

         --Signals used for Testing Only
        Timer_Out:      out unsigned(log2(8+1)-1 downto 0);
        Timer_Next_Out: out unsigned(log2(8+1)-1 downto 0);
        M_ACKNOWLEDGE_L_test_Out:       out std_logic
    );
end MBUS_Controller;

architecture MBUS_Controller_arch of MBUS_Controller is

--constants

constant Mem_Blk_1_L : natural := 1048576 ;
--Lower bound of VPM Master OBM (100000H)
constant Mem_Blk_1_H : natural := 2097151 ;
--Upper bound of VPM Master OBM (1FFFFFH)
constant Mem_Blk_1_Up_Bits : unsigned(2 downto 0) := "001";
--Bits 22-20 of Address = 001 if in Blk 1
constant Mem_Blk_2_L : natural := 2097152 ;
--Lower bound of VPM Slave1 OBM (200000H)
constant Mem_Blk_2_H : natural := 3145727 ;
--Upper bound of VPM Slave1 OBM (2FFFFFH)

constant MAX_DELAY:     natural := 200;
-- Max Delay interval (ns) (Changed for testing only)
constant TIMER_CYCLES:  natural := 1 + ((MAX_DELAY * FREQ) / 1000000);
-- ACK Signal Max Delay (20ns)

--Constants for Clarity of Code
constant ACTIVE: std_logic := '1';
```

```
constant ACTIVE_L: std_logic := '0';          --For active low signal
constant INACTIVE: std_logic := '0';
constant INACTIVE_L: std_logic := '1';        --For active low signal

signal Timer, Timer_next: unsigned(log2(TIMER_CYCLES+1)-1 downto 0);
-- current Delay time

--All signals tied to input/output have same name with _Int addended

signal  Clk_Int :         std_logic;
signal  Rst_Int :         std_logic;
signal  P_Grant_Int:   std_logic;       --Signal used for Processor Grant Indication
signal  M_BUS_Int:   unsigned(22 downto 0);                 --INOUT
signal  M_BUS_Read:          unsigned(22 downto 0);
signal  P_Data_WR_Int:       unsigned(31 downto 0);
signal  P_Data_RD_Int:       unsigned(31 downto 0);
signal  P_Addr_Int:    unsigned(22 downto 0);
signal  P_RD_Req_Int:        std_logic;
signal  P_WR_Req_Int:        std_logic;
--Signals used for Grant_Logic
signal  M_GRANT_OUT_Int:          unsigned(1 downto 0);
signal  M_Grant_Proc_Int:    std_logic;
--Signals used for control logic
signal  M_DESIRE_IN_L_Int:                 unsigned(1 downto 0);
signal  M_REQUEST_L_Int: std_logic;             --INOUT
signal  M_ACKNOWLEDGE_L_Int:          std_logic;
signal  M_RESUME_L_Int:  std_logic;             --INOUT
signal  MSB_WRITE_L_Int: std_logic;             --INOUT
signal  LSB_WRITE_L_Int:  std_logic;            --INOUT
signal  THREE_TWO_DATA_Int:   std_logic;                --INOUT
signal  IPL_WRITE_Int:                std_logic;                --INOUT
signal  M_BUSY_L_Int:                 std_logic;                --INOUT
signal  Mem_DONE_Int:                 std_logic;
signal  S_BUSY_L_Int:                 std_logic;
signal  Mem_Addr_Int:                 unsigned(22 downto 0);
signal  BUS_ERROR_L_Int: std_logic;
--Signal used for timeout
signal  Time_Out:      std_logic;
--Signal to indicate Parity Error
signal  Parity_Error_Int:        std_logic;
--Signals for parity generation for External drivers of signals
signal  LSB_Parity_Generate_Input:           std_logic;
signal  MSB_Parity_Generate_Input:           std_logic;
signal  ADRS_Parity_Generate_Input:                  std_logic;
signal  CMD_Parity_Generate_Input:                   std_logic;
--Signals for parity generation for Internal drivers of signals
```

```
signal  LSB_Parity_Generate_Output:                std_logic;
signal  MSB_Parity_Generate_Output:                std_logic;
signal  ADRS_Parity_Generate_Output:               std_logic;
signal  CMD_Parity_Generate_Output:                std_logic;
--Signals for parity input
signal  LSB_Parity_Int:             std_logic;     --INOUT
signal  MSB_Parity_Int:             std_logic;     --INOUT
signal  ADRS_Parity_Int:            std_logic;     --INOUT
signal  CMD_Parity_Int:             std_logic;     --INOUT
--Signal for Parity Generator Format
signal  ADRS_Parity_Input:  unsigned(7 downto 0);
signal  ADRS_Parity_Output:unsigned(7 downto 0);
signal  CMD_Parity_Input:   unsigned(7 downto 0);
signal  CMD_Parity_Output: unsigned(7 downto 0);

--Signal to drive INOUTS
signal Drive_MBUS:          std_logic;
signal Drive_Resume:        std_logic;
signal Drive_Request:       std_logic;
signal Drive_M_Busy:         std_logic;
signal Drive_Bus_Error:     std_logic;
signal Drive_LSB_Parity:    std_logic;
signal Drive_MSB_Parity:    std_logic;
signal Drive_ADRS_Parity:   std_logic;
signal Drive_CMD_Parity:    std_logic;
signal Drive_MSB_Write:             std_logic;
signal Drive_LSB_Write:             std_logic;
signal Drive_Three_Two_Data:        std_logic;
signal Drive_IPL_Write:             std_logic;

--Signals to Latch
signal M_ACKNOWLEDGE_L_test:std_logic;
signal Mem_Data_RD_Int:             unsigned(31 downto 0);
signal Mem_Data_WR_Int:             unsigned(31 downto 0);
signal Mem_Data_WR_Int_Out:         unsigned(31 downto 0);
--Latch Driver Signals
signal M_ACK_Latch:                 std_logic;
signal P_DATA_RD_Latch:             std_logic;
signal M_Addr_Latch:                std_logic;
signal Mem_Data_RD_Latch:           std_logic;
signal Mem_Data_WR_Latch:           std_logic;
```

```
type FSM_type is
        (Idle, Addr_Out_M,Req_M, Ack_Read_M, Data_Clk_In_M, Rsm_Read_M,
        Ack_Write_M, Data_Clk_Out_M, Rsm_Write_M,
        Req_Read_S,AddClkIn_Read_S, Ack_Read_S, Rsm_Read_S, Read_Done_S,
        Req_Write_S, AddClkIn_Write_S, Ack_Write_S, Write_Data_S, Rsm_Write_S,
        Write_Done_S,Error_Internal, Error_External);
                --Req_M - if Master has use of MBUS
                --Req_Write_S - if slave has use of MBUS for Write Operation
                --Req_Read_S - if slave has use of MBUS for Read Operation
                --Ack_Read_M - Acknowlege Phase of a Master read operation
                --Data_Clk_In_M - State that clocks in Data off BUS
                --Ack_Write_M - Acknowlege Phase of a Master write operation
                --Ack_Read_S - Acknowlege Phase of a Slave read operation
                --Ack_Write_S - Acknowlege Phase of a Slave write operation
                --Rsm_Read_M - Resume Phase of a Master read operation
                --Rsm_Write_M     - Resume Phase of a Master write operation
                --Data_Clk_Out_M - Clock Out the Data to be written
                --Rsm_Write_S - Resume Phase of a slave read operation
                --Rsm_Read_S - Resume Phase of a slave write operation
                --Error_Internal- Error state caused by Internal Error
                --Error_External- Error state caused by External Error
                --AddCLkIn_Read_S- Clock in Address for Read operation
                --AddClkIn_Write_S- Clock in Address for Write operation
                --Read_Done_S - Data removed from bus but bus not available yet
                --DataClkIn_Write_S - Clock in data to write to memory
                --Write_Data_S - Wait state for data to be written to memory
                --Write_Done_S - Wait state for completion of Write operation
                --Addr_Out_M - Wait 1 clock after puting address on Bus to
                        --drive Request Signal

signal Curr_State, Next_State : FSM_Type;


begin

--Connect all appropriate signals
Clk_Int <= Clk;
Rst_Int <= Rst;
M_GRANT_OUT <= M_GRANT_OUT_Int;
--Connect Grant signals to output port
--P_RD_Req_Int <= P_RD_Req;
--P_WR_Req_Int <= P_WR_Req;
P_Addr_Int <= P_Addr;
P_Data_WR_Int <= P_Data_WR;
P_Grant_Out <= M_Grant_Proc_Int;
M_DESIRE_IN_L_Int <= M_DESIRE_IN_L;
```

```
M_ACKNOWLEDGE_L_Int <= M_ACKNOWLEDGE_L;
S_BUSY_L <= S_BUSY_L_Int;
Mem_Data_WR <= Mem_Data_WR_Int_Out;
Mem_DONE_Int <= Mem_DONE;
M_BUS_Read <= M_BUS;
Mem_Addr <= Mem_Addr_Int;

--Tristates for INOUTs

M_RESUME_L <= M_RESUME_L_Int      when Drive_Resume = ACTIVE else ('Z');
M_BUS <= M_BUS_Int                when Drive_MBUS = ACTIVE else (others =>'Z');
M_REQUEST_L <= M_REQUEST_L_Int    when Drive_Request = ACTIVE else ('Z');
M_BUSY_L <= M_BUSY_L_Int          when Drive_M_Busy = ACTIVE else ('Z');
LSB_PARITY <= LSB_PARITY_Int when Drive_LSB_Parity = ACTIVE else ('Z');
MSB_PARITY <= MSB_PARITY_Int when Drive_MSB_Parity = ACTIVE else ('Z');
ADRS_PARITY <= ADRS_PARITY_Int when Drive_ADRS_Parity = ACTIVE else
('Z');
CMD_PARITY <= CMD_PARITY_Int when Drive_CMD_Parity = ACTIVE else ('Z');
MSB_WRITE_L <= MSB_WRITE_L_Int   when Drive_MSB_Write = ACTIVE else
('Z');
LSB_WRITE_L <= LSB_WRITE_L_Int when Drive_LSB_Write = ACTIVE else ('Z');
THREE_TWO_DATA <= THREE_TWO_DATA_Int when Drive_Three_Two_Data =
ACTIVE else ('Z');
IPL_WRITE <= IPL_WRITE_Int        when Drive_IPL_Write = ACTIVE else ('Z');
BUS_ERROR_L <= BUS_ERROR_L_Int   when Drive_Bus_Error = ACTIVE else
('Z');

--Latch Signals
P_DATA_RD_Int <= ("0000000000000000" & M_Bus(15 downto 0)) when
P_DATA_RD_Latch = ACTIVE else P_DATA_RD_Int;
P_Data_RD <= P_Data_RD_Int;
Mem_Addr_Int <= M_BUS when M_Addr_Latch = ACTIVE else Mem_Addr_Int;
Mem_Data_RD_Int <= Mem_Data_RD when Mem_Data_RD_Latch = ACTIVE else
Mem_Data_RD_Int;
Mem_Data_WR_Int_Out <= Mem_Data_WR_Int when Mem_Data_WR_Latch =
ACTIVE else Mem_Data_WR_Int_Out;

--Signals for Testing only
Timer_Out <= Timer;
Timer_Next_Out <= Timer_Next;
--Latch Test
M_ACKNOWLEDGE_L_test <= M_ACKNOWLEDGE_L when M_ACK_Latch =
ACTIVE else M_ACKNOWLEDGE_L_test;
M_ACKNOWLEDGE_L_test_Out <= M_ACKNOWLEDGE_L_test;

--Assigning Signals for Parity Generator
```

```
ADRS_Parity_Input <= M_BUS_Int(22 downto 16) & "0";
ADRS_Parity_Output <= P_Addr_Int(22 downto 16) & "0";
CMD_Parity_Input <= MSB_WRITE_L & LSB_WRITE_L & THREE_TWO_DATA &
IPL_WRITE & "0000";
CMD_Parity_Output <= MSB_WRITE_L_Int & LSB_WRITE_L_Int &
THREE_TWO_DATA_Int & IPL_WRITE_Int & "0000";

--Instantiate Grant Logic Module
 u0: Grant_logic port map (   M_Desire_Ext => M_DESIRE_IN_L_Int,
                              M_Desire_Proc => P_Desire_L,
                              M_Grant_Ext => M_GRANT_OUT_Int  ,
                               --Grant Signal to external signal
                              M_Grant_Proc => M_Grant_Proc_Int ,
                               --Grant Signal to internal signal
                               Clk => Clk_Int,
                              Rst => Rst_Int
                          );

--Instantiate Parity Generator
 --LSB Parity for Input
 u1: oddParityGen port map (
      data => M_BUS_Int(7 downto 0),
      parity => LSB_Parity_Generate_Input
   );

 u2: oddParityGen port map (
      data => P_Addr_Int(7 downto 0),
      parity => LSB_Parity_Generate_Output
   );
 --MSB Parity for Input
 u3: oddParityGen port map (
      data => M_BUS_Int(15 downto 8),
      parity => MSB_Parity_Generate_Input
   );

 u4: oddParityGen port map (
      data => P_Addr_Int(15 downto 8),
      parity => MSB_Parity_Generate_Output
   );
 --ADRS Parity for Input
 u5: oddParityGen port map (
      data => ADRS_Parity_Input,
      parity => ADRS_Parity_Generate_Input
   );

 u6: oddParityGen port map (
```

```
        data => ADRS_Parity_Input,
        parity => ADRS_Parity_Generate_Output
    );
 --CMD Parity for Input
  u7: oddParityGen port map (
        data => CMD_Parity_Input,
        parity => CMD_Parity_Generate_Input
    );

  --CMD Parity for Output
  u8: oddParityGen port map (
        data => CMD_Parity_Output,
        parity => CMD_Parity_Generate_Output
    );



--Next State Conditioning Logic (Process 1)

nxtStProc: process(Curr_State,Timer,Timer_next,BUS_ERROR_L,M_DESIRE_IN_L,
                   Mem_DONE,LSB_WRITE_L,CMD_Parity,
                   M_RESUME_L,M_Grant_Proc_Int,MSB_Parity_Generate_Input,LSB_Pa
                   rity_Generate_Input,M_BUSY_L,M_BUS_Read,M_BUS_Int,M_BUS,
                   MSB_WRITE_L,Time_Out,P_RD_Req,CMD_Parity_Generate_Input,P_
                   WR_Req,MSB_Parity,
                   ADRS_Parity_Generate_Input,M_REQUEST_L,M_ACKNOWLEDGE_
                   L_Int,M_GRANT_OUT_Int,LSB_Parity,
                   ADRS_Parity,M_DESIRE_IN_L_Int)

  begin

      case Curr_State is

        when Idle =>
        --Go to Master states if processor has been granted bus use
          if M_Grant_Proc_Int = ACTIVE then
            next_state <=  Addr_Out_M;
            Timer_Next <= TO_UNSIGNED(TIMER_CYCLES,Timer'length);
             --Start Timer
        --If Slave has bus use AND address in OBM range AND Write signals are active
GOTO Slave Write states
          elsif ((M_GRANT_OUT_Int(0) = ACTIVE or M_GRANT_OUT_Int(1) =
ACTIVE) and M_REQUEST_L = ACTIVE_L and (M_BUS(22 downto 20) =
Mem_Blk_1_Up_Bits) and MSB_WRITE_L = ACTIVE_L
                and LSB_WRITE_L = ACTIVE_L) then
            --Check Parity
```

```
            if (LSB_Parity_Generate_Input = LSB_Parity and MSB_Parity_Generate_Input =
MSB_Parity and CMD_Parity_Generate_Input = CMD_Parity and
ADRS_Parity_Generate_Input = ADRS_Parity) then
              next_state <=  Req_Write_S;
             Timer_Next<= TO_UNSIGNED(TIMER_CYCLES,Timer'length); --Start Timer
            else
              next_state <= Error_Internal;
            end if;
        --If Slave has bus use AND address in OBM range AND Write signals are
INACTIVE GOTO Slave Write states
          elsif ((M_GRANT_OUT_Int(0) = ACTIVE or M_GRANT_OUT_Int(1) =
ACTIVE) and M_REQUEST_L = ACTIVE_L and (M_BUS(22 downto 20) =
Mem_Blk_1_Up_Bits) and MSB_WRITE_L = INACTIVE_L and LSB_WRITE_L =
INACTIVE_L) then
            --Check Parity
            if (LSB_Parity_Generate_Input = LSB_Parity and MSB_Parity_Generate_Input =
MSB_Parity and CMD_Parity_Generate_Input = CMD_Parity and
ADRS_Parity_Generate_Input = ADRS_Parity) then
              next_state <=  Req_Read_S;
             Timer_Next<= TO_UNSIGNED(TIMER_CYCLES,Timer'length); --Start Timer
            else
              next_state <= Error_Internal;
            end if;
         else
           next_state <= Idle;
         end if;
--States for Master Bus Usage
        when Addr_Out_M =>
           next_state <= Req_M;

        when Req_M =>
           if (M_ACKNOWLEDGE_L_Int = ACTIVE_L) then
             if (P_RD_Req = ACTIVE) then
               next_state <= Ack_Read_M;
             elsif (P_WR_Req = ACTIVE) then
               next_state <= Ack_Write_M;
             end if;
           elsif (BUS_ERROR_L = ACTIVE_L) then
             next_state <= Error_External;
             elsif (Time_Out = ACTIVE and M_ACKNOWLEDGE_L_Int = INACTIVE_L)
             then
             next_state <= Error_Internal;
           else
             next_state <= Req_M;
           end if;
            --States for Master Read
```

```
        when Ack_Read_M =>
          --if (M_RESUME_L_Int = ACTIVE_L) then
          if (M_RESUME_L = ACTIVE_L) then
            next_state <= Data_Clk_In_M;
          else
            next_state <= Ack_Read_M;
          end if;

        when Data_Clk_In_M =>
          next_state <= Rsm_Read_M;

        when Rsm_Read_M =>

          if (M_ACKNOWLEDGE_L_Int = INACTIVE_L) then
            next_state <= Idle;
          else
            next_state <= Rsm_Read_M;
          end if;

        --States for Master Write
        when Ack_Write_M =>
           next_state <= Data_Clk_Out_M;

        when Data_Clk_Out_M =>

          if (M_RESUME_L = ACTIVE_L) then
            next_state <= Rsm_Write_M;
          else
            next_state <= Data_Clk_Out_M;
          end if;

        when Rsm_Write_M =>

          if (M_ACKNOWLEDGE_L_Int = INACTIVE_L) then
            next_state <= Idle;
          else
            next_state <= Rsm_Write_M;
          end if;

--States for External user of MBUS

--States for a Slave Read
        when Req_Read_S =>
          next_state <= AddCLkIn_Read_S;

        when AddCLkIn_Read_S =>
```

```
        if (Mem_DONE = ACTIVE) then
          next_state <= Ack_Read_S;
        else
          next_state <= AddCLkIn_Read_S;
        end if;

      when Ack_Read_S =>
        if (M_REQUEST_L = INACTIVE_L) then
          next_state <= Rsm_Read_S;
        else
          next_state <= Ack_Read_S;
        end if;

      when Rsm_Read_S =>
        if (M_BUSY_L = INACTIVE_L) then
          next_state <= Read_Done_S;
        else
          next_state <= Rsm_Read_S;
        end if;

      when Read_Done_S =>
        next_state <= Idle;

--States for Slave Write

      when Req_Write_S =>
        next_state <= AddClkIn_Write_S;

      when AddClkIn_Write_S =>
        if (M_REQUEST_L = INACTIVE_L) then
          next_state <= Ack_Write_S;
        else
          next_state <= AddClkIn_Write_S;
        end if;

      when Ack_Write_S =>
        next_state <= Write_Data_S;

      when Write_Data_S =>
        if (Mem_DONE = ACTIVE) then
          next_state <= Rsm_Write_S;
        else
          next_state <= Write_Data_S;
        end if;

      when Rsm_Write_S =>
```

```
          if (M_BUSY_L = INACTIVE_L) then
            next_state <= Write_Done_S;
          else
            next_state <= Rsm_Write_S;
          end if;

        when Write_Done_S =>
          next_state <= Idle;


-- States for errors
        when Error_Internal =>
          if ((M_DESIRE_IN_L(0) = INACTIVE_L and M_GRANT_OUT_Int(0) =
INACTIVE) or (M_DESIRE_IN_L(1) = INACTIVE_L and M_GRANT_OUT_Int(1) =
INACTIVE)) then
             next_state <= Idle;
          else
            next_state <= Error_Internal;
          end if;

            when Error_External =>
              if (BUS_ERROR_L = INACTIVE_L) then
                next_state <= Idle;
              else
                next_state <= Error_External;
              end if;

            when others =>
              null;

       end case;

       --Timer will count down after being started by leaving Idle State
     case Curr_State is

       when Idle =>
         null;

       when others =>

         if Timer /= TO_UNSIGNED(0,Timer'length) then
           Timer_next <= Timer - 1;
           Time_Out <= INACTIVE;
         else
           --Timer_next <= Timer;
           Time_Out <= ACTIVE;
```

```
        end if;
      end case;

    end process nxtStProc;

--Current State Vector Register (Process 2)

  curStProc: process (Clk_Int, Rst_Int)
  begin
    if (Rst_Int = '0') then
       Curr_State <= Idle;
    elsif (Clk_Int'event and Clk_Int ='1') then
          Curr_State <= Next_State;
          Timer <= Timer_next;
    end if;
  end process curStProc;


--Output Conditioning Logic (Process 3)
  outConProc:
process(Curr_State,Mem_Data_RD_Int,MSB_Parity_Generate_Input,M_BUS_Int,LSB_
Parity_Generate_Input,P_RD_Req,CMD_Parity_Generate_Input,ADRS_Parity_Generate
_Input,P_Data_WR_Int,P_Addr_Int,LSB_Parity_Generate_Output,MSB_Parity_Generat
e_Output,ADRS_Parity_Generate_Output, CMD_Parity_Generate_Output,M_BUS)

  begin
          --Default Signal to drive all Tristates High Z
          Drive_MBUS <= INACTIVE;
          M_RESUME_L_Int <= INACTIVE_L;
          Drive_Resume <= INACTIVE;
          M_REQUEST_L_Int <= INACTIVE_L;
          Drive_Request <= INACTIVE;
          M_BUSY_L_Int <= INACTIVE_L;
          Drive_M_Busy <= INACTIVE;
          BUS_ERROR_L_Int <= INACTIVE;
          Drive_Bus_Error <= INACTIVE;
          LSB_PARITY_Int <= INACTIVE;
          Drive_LSB_Parity <= INACTIVE;
          MSB_PARITY_Int <= INACTIVE;
          Drive_MSB_Parity <= INACTIVE;
          ADRS_PARITY_Int <= INACTIVE;
          Drive_ADRS_Parity <= INACTIVE;
          CMD_PARITY_Int <= INACTIVE;
          Drive_CMD_Parity <= INACTIVE;
          MSB_WRITE_L_Int <= INACTIVE;
          Drive_MSB_Write <= INACTIVE;
```

```
        LSB_WRITE_L_Int <= INACTIVE;
        Drive_LSB_Write <= INACTIVE;
        THREE_TWO_DATA_Int <= INACTIVE;
        Drive_Three_Two_Data <= INACTIVE;
        IPL_WRITE_Int <= INACTIVE;
        Drive_IPL_Write <= INACTIVE;
      --Drive all outs inactive
      S_BUSY_L_Int <= INACTIVE_L;
      P_Mem_Done <= INACTIVE;
      Mem_WR_Req <= INACTIVE;
      Mem_RD_Req <= INACTIVE;
        --Latch Drivers
        M_ACK_Latch <= INACTIVE;
        P_DATA_RD_Latch <= INACTIVE;
        Mem_Data_RD_Latch <= INACTIVE;
        Mem_Data_WR_Latch <= INACTIVE;
        M_Addr_Latch <= INACTIVE;


   case Curr_State is

    when Idle =>
        null;

--States for Master Operations

    when Addr_Out_M =>
     M_BUS_Int <= P_Addr_Int;       --Put Address on Bus
     Drive_MBUS <= ACTIVE;
     --Command Signals
     Drive_MSB_Write <= ACTIVE;
     Drive_LSB_Write <= ACTIVE;
     MSB_WRITE_L_Int <= P_RD_Req;
     LSB_WRITE_L_Int <= P_RD_Req;
--This signal is active low. The RD signal is active high,therfore
--when the write signal is active, the read signal will be low.
     Drive_Three_Two_Data <= ACTIVE;
     THREE_TWO_DATA_Int <= INACTIVE;
     Drive_IPL_Write <= ACTIVE;
     IPL_WRITE_Int <= INACTIVE;
        --Assign Parity Values
     Drive_MSB_Parity <= ACTIVE;
     MSB_PARITY_Int <= MSB_Parity_Generate_Output;
     Drive_LSB_Parity <= ACTIVE;
     LSB_PARITY_Int <= LSB_Parity_Generate_Output;
     Drive_ADRS_Parity <= ACTIVE;
```

```
      ADRS_PARITY_Int <= ADRS_Parity_Generate_Output;
      Drive_CMD_Parity <= ACTIVE;
      CMD_PARITY_Int <= CMD_Parity_Generate_Output;

    when Req_M =>
      --Bus Control Signals
      M_BUS_Int <= P_Addr_Int;      --Put Address on Bus
      Drive_MBUS <= ACTIVE;

      Drive_Request <= ACTIVE;
      M_REQUEST_L_Int <= ACTIVE_L;
--Drive the control signal low to indicate Address is valid
      Drive_MSB_Parity <= ACTIVE;
      MSB_PARITY_Int <= MSB_Parity_Generate_Output;
      Drive_LSB_Parity <= ACTIVE;
      LSB_PARITY_Int <= LSB_Parity_Generate_Output;
      Drive_ADRS_Parity <= ACTIVE;
      ADRS_PARITY_Int <= ADRS_Parity_Generate_Output;
      Drive_CMD_Parity <= ACTIVE;
      CMD_PARITY_Int <= CMD_Parity_Generate_Output;
      Drive_MSB_Write <= ACTIVE;
      Drive_LSB_Write <= ACTIVE;
      MSB_WRITE_L_Int <= P_RD_Req;
      LSB_WRITE_L_Int <= P_RD_Req;
--This signal is active low. The RD signal is active high,therfore
--when the write signal is active, the read signal will be low.
      Drive_Three_Two_Data <= ACTIVE;
      THREE_TWO_DATA_Int <= INACTIVE;
      Drive_IPL_Write <= ACTIVE;
      IPL_WRITE_Int <= INACTIVE;

      M_ACK_Latch <= ACTIVE;

--State for Master Read
    when Ack_Read_M =>
      --Activate M_Busy Signal
      Drive_M_Busy <= ACTIVE;
      M_BUSY_L_Int <= ACTIVE_L;

    when Data_Clk_In_M =>
      P_Data_RD_Latch <= ACTIVE;
      Drive_M_Busy <= ACTIVE;
      M_BUSY_L_Int <= ACTIVE_L;

    when Rsm_Read_M =>
      P_Mem_Done <= ACTIVE;
```

```
      Drive_M_Busy <= ACTIVE;
      M_BUSY_L_Int <= INACTIVE_L;

--States for Master Write
    when Ack_Write_M =>
      Drive_Request <= ACTIVE;
      M_REQUEST_L_Int <= ACTIVE_L;
      Drive_MSB_Write <= ACTIVE;
      Drive_LSB_Write <= ACTIVE;
      MSB_WRITE_L_Int <= P_RD_Req;
      LSB_WRITE_L_Int <= P_RD_Req;
      Drive_MSB_Parity <= ACTIVE;
      MSB_PARITY_Int <= MSB_Parity_Generate_Output;
      Drive_LSB_Parity <= ACTIVE;
      LSB_PARITY_Int <= LSB_Parity_Generate_Output;

      Drive_M_Busy <= ACTIVE;
      M_BUSY_L_Int <= ACTIVE_L;
      --Drve the MBUS with data
      Drive_MBUS <= ACTIVE;
      M_BUS_Int <= ("0000000" & P_Data_WR_Int(15 downto 0));

    when Data_Clk_Out_M =>
      Drive_MSB_Parity <= ACTIVE;
      Drive_MSB_Write <= ACTIVE;
      Drive_LSB_Write <= ACTIVE;
      MSB_WRITE_L_Int <= P_RD_Req;
      LSB_WRITE_L_Int <= P_RD_Req;
      MSB_PARITY_Int <= MSB_Parity_Generate_Output;
      Drive_LSB_Parity <= ACTIVE;
      LSB_PARITY_Int <= LSB_Parity_Generate_Output;
      Drive_M_Busy <= ACTIVE;
      M_BUSY_L_Int <= ACTIVE_L;
      Drive_MBUS <= ACTIVE;
      M_BUS_Int <= ("0000000" & P_Data_WR_Int(15 downto 0));

    when Rsm_Write_M =>
      P_Mem_Done <= ACTIVE;
      Drive_MSB_Write <= ACTIVE;
      MSB_WRITE_L_Int <= INACTIVE_L;
      Drive_LSB_Write <= ACTIVE;
      LSB_WRITE_L_Int <= INACTIVE_L;
      Drive_M_Busy <= ACTIVE;
      M_BUSY_L_Int <= INACTIVE_L;
      --M_BUS_Int <= (others => 'Z');
```

```
--States for External user of MBUS

--States for Slave Read
    when Req_Read_S =>
     M_Addr_Latch <= ACTIVE;
     Mem_RD_Req <= ACTIVE;

    when AddCLkIn_Read_S =>
     Mem_RD_Req <= ACTIVE;
     Mem_Data_RD_Latch <= ACTIVE;--Latches Data off of SDRAM
     S_BUSY_L_Int <= ACTIVE_L; --Notify user that address is clocked in


    when Ack_Read_S =>
     Drive_MBUS <= ACTIVE;
     M_BUS_Int <= ("0000000" & Mem_Data_RD_Int(15 downto 0));
     Drive_MSB_Parity <= ACTIVE;
     MSB_PARITY_Int <= MSB_Parity_Generate_Output;
     Drive_LSB_Parity <= ACTIVE;
     LSB_PARITY_Int <= LSB_Parity_Generate_Output;
     S_BUSY_L_Int <= ACTIVE_L;


    when Rsm_Read_S =>
     Drive_MBUS <= ACTIVE;
     M_BUS_Int <= ("0000000" & Mem_Data_RD_Int(15 downto 0));
     S_BUSY_L_Int <= ACTIVE_L;
     M_RESUME_L_Int <= ACTIVE_L;
     Drive_Resume <= ACTIVE;


    when Read_Done_S =>
     M_RESUME_L_Int <= ACTIVE_L;
     Drive_Resume <= ACTIVE;
     S_BUSY_L_Int <= ACTIVE_L;

--States for Slave Write

    when Req_Write_S =>
     M_Addr_Latch <= ACTIVE;
     Drive_Resume <= ACTIVE;

    when AddClkIn_Write_S =>
     S_BUSY_L_Int <= ACTIVE_L;
     Drive_Resume <= ACTIVE;
```

```
    when Ack_Write_S =>
      Mem_Data_WR_Latch <= ACTIVE;
      Mem_Data_WR_Int <= ("0000000000000000" & M_BUS(15 downto 0));
      Mem_WR_Req <= ACTIVE;
      S_BUSY_L_Int <= ACTIVE_L;
      Drive_Resume <= ACTIVE;

    when Write_Data_S =>
      S_BUSY_L_Int <= ACTIVE_L;
      Drive_Resume <= ACTIVE;
      Mem_WR_Req <= ACTIVE;

    when Rsm_Write_S =>
      M_RESUME_L_Int <= ACTIVE_L;
      Drive_Resume <= ACTIVE;
      Mem_WR_Req <= INACTIVE;
      S_BUSY_L_Int <= ACTIVE_L;

    when Write_Done_S =>
      M_RESUME_L_Int <= ACTIVE_L;
      Drive_Resume <= ACTIVE;

--States for Errors
    when Error_Internal =>
      null;

    when Error_External =>
      Drive_Bus_Error <= ACTIVE;
      BUS_ERROR_L_Int <= ACTIVE_L;

    when others =>
      null;

   end case;

  end process outConProc;

end MBUS_Controller_arch;
```

===
XBUS Arbitrator <x_grant_logic.vhd>
===

Project:	AYK-14 VHSIC Processor Module Hardware Emulator
Component:	XBUS Arbitrator
Description:	State Machine that determines the next user of the XBUS via a rotating priority scheme and generates the control signals to notify the current user. The signals monitored are the Desire signals from 6 external users plus the Processor. The control signals generated are the Grant Signals.

-- Author: LT Bryan Fetter, USN
-- Advisor: Dr. Russ Duren
-- Co-advisor: Dr. Hersch Loomis
-- Location: Naval Postgraduate School

-- Created: 25 October 2002
-- Modified: 21 November 2002
-- Simulated:
-- Target: XCV1000E FG1156
-- Software: Foundation 4.2i
-- Notes:

Disclaimer: NPS, makes no warranty for the use of this code or design. This code is provided "As Is". NPS, assumes no responsibility for any errors, which may appear in this code, nor does it make a commitment to update the information contained herein. NPS specifically disclaims any implied warranties of fitness for a particular purpose.

===

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;

package X_GRANT is

component X_GRANT_LOGIC
   port (
      X_Desire: in std_logic_vector (6 downto 0);
      X_Grant: out std_logic_vector (6 downto 0);
      X_Resume: inout STD_LOGIC;
```

```
        Clk: in STD_LOGIC;
        Rst: in STD_LOGIC
    );
end component;

end package X_GRANT;

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;

entity X_GRANT_LOGIC is
    port (
        X_Desire: in std_logic_vector (6 downto 0);
        X_Grant: out std_logic_vector (6 downto 0);
        X_Resume: inout STD_LOGIC;
        Clk: in STD_LOGIC;
        Rst: in STD_LOGIC
    );
end X_GRANT_LOGIC;

architecture X_GRANT_LOGIC_arch of X_GRANT_LOGIC is

type FSM_type is (Idle,Grant);
signal Curr_State, Next_State : FSM_Type;
signal Next_User : std_logic_vector (2 downto 0);
signal Pri_0,Pri_1,Pri_2,Pri_3,Pri_4,Pri_5,Pri_6 : std_logic_vector (2 downto 0);

signal X_Desire_Int : std_logic_vector (6 downto 0);
signal X_Grant_Int : std_logic_vector (6 downto 0);
signal X_Resume_Int: std_logic;


begin

X_Desire_Int <= X_Desire;
X_Resume_Int <= X_Resume;
X_Grant <= X_Grant_Int;


nxtStProc: process(Curr_State,Next_State,
                   X_Desire_Int, X_Resume_Int,Next_User)

  begin
```

```
    case Curr_State is

      when Idle =>
        if X_Desire_Int /= "1111111" then
          Next_State <= Grant;
        else
          Next_State <= Idle;
        end if;

      when Grant =>

            if (X_Resume_Int = '1'
                and X_Desire_Int(conv_integer(Next_User)) = '1') then
              Next_State <= Idle;
            else
              Next_State <= Grant;
            end if;

          when others =>
            null;

     end case;
   end process nxtStProc;

--Process to register current state

  curStProc: process (Clk, Rst)
  begin
    if (Rst = '0') then
       Curr_State <= Idle;
    elsif (Clk'event and Clk ='1') then
       Curr_State <= Next_State;
    end if;
  end process curStProc;

--Process to generate outputs

  outConProc: process(Curr_State,X_Desire_Int,Pri_0,Pri_1,Pri_2,
                            Pri_3,Pri_4,Pri_5,Pri_6,Next_User)

  begin

    case Curr_State is

     when Idle =>
```

```
  X_Grant_Int <= "0000000";

  --The 1st If statement is to handle the reset case

  if (Pri_0 = Pri_1) then
    if (X_Desire_Int(conv_integer(0)) = '0' )then
      Next_User <= "000";
    elsif (X_Desire_Int(conv_integer(1)) = '0' )then
      Next_User <= "001";
    elsif (X_Desire_Int(conv_integer(2)) = '0' )then
      Next_User <= "010";
    elsif (X_Desire_Int(conv_integer(3)) = '0' )then
      Next_User <= "011";
    elsif (X_Desire_Int(conv_integer(4)) = '0' )then
      Next_User <= "100";
    elsif (X_Desire_Int(conv_integer(5)) = '0' )then
      Next_User <= "101";
    elsif (X_Desire_Int(conv_integer(6)) = '0' )then
    end if;

  elsif X_Desire_Int(conv_integer(Pri_0)) = '0'then
    Next_User <= Pri_0;
  elsif X_Desire_Int(conv_integer(Pri_1)) = '0'then
    Next_User <= Pri_1;
  elsif X_Desire_Int(conv_integer(Pri_2)) = '0'then
    Next_User <= Pri_2;
  elsif X_Desire_Int(conv_integer(Pri_3)) = '0'then
    Next_User <= Pri_3;
  elsif X_Desire_Int(conv_integer(Pri_4)) = '0'then
    Next_User <= Pri_4;
  elsif X_Desire_Int(conv_integer(Pri_5)) = '0'then
    Next_User <= Pri_5;
  elsif X_Desire_Int(conv_integer(Pri_6)) = '0'then
    Next_User <= Pri_6;
  end if;


when Grant =>
  X_Grant_Int(conv_integer(Next_User)) <= '1';

  if Next_User = "000" then
    Pri_0 <= "001";
    Pri_1 <= "010";
    Pri_2 <= "011";
    Pri_3 <= "100";
    Pri_4 <= "101";
```

```
  Pri_5 <= "110";
  Pri_6 <= "000";
elsif Next_User = "001" then
  Pri_0 <= "010";
  Pri_1 <= "011";
  Pri_2 <= "100";
  Pri_3 <= "101";
  Pri_4 <= "110";
  Pri_5 <= "000";
  Pri_6 <= "001";
elsif Next_User = "010" then
  Pri_0 <= "011";
  Pri_1 <= "100";
  Pri_2 <= "101";
  Pri_3 <= "110";
  Pri_4 <= "000";
  Pri_5 <= "001";
  Pri_6 <= "010";
elsif Next_User = "011" then
  Pri_0 <= "100";
  Pri_1 <= "101";
  Pri_2 <= "110";
  Pri_3 <= "000";
  Pri_4 <= "001";
  Pri_5 <= "010";
  Pri_6 <= "011";
elsif Next_User = "100" then
  Pri_0 <= "101";
  Pri_1 <= "110";
  Pri_2 <= "000";
  Pri_3 <= "001";
  Pri_4 <= "010";
  Pri_5 <= "011";
  Pri_6 <= "100";
elsif Next_User = "101" then
  Pri_0 <= "110";
  Pri_1 <= "000";
  Pri_2 <= "001";
  Pri_3 <= "010";
  Pri_4 <= "011";
  Pri_5 <= "100";
  Pri_6 <= "101";
elsif Next_User = "110" then
  Pri_0 <= "000";
  Pri_1 <= "001";
  Pri_2 <= "010";
```

```
        Pri_3 <= "011";
        Pri_4 <= "100";
        Pri_5 <= "101";
        Pri_6 <= "110";
      else
        Pri_0 <= "001";
        Pri_1 <= "010";
        Pri_2 <= "011";
        Pri_3 <= "100";
        Pri_4 <= "101";
        Pri_5 <= "110";
        Pri_6 <= "000";
      end if;

    when others =>
      null;

   end case;

  end process outConProc;

end X_GRANT_LOGIC_arch;
```

==

MBUS Desire / Grant Arbitrator <grant_logic.vhd>

==

Project: AYK-14 VHSIC Processor Module Hardware Emulator
Component: MBUS Grant Arbitrator

Description: State machine that provides rotating priority logic to determinethe next user of the MBUS. The component analyzes the MBUS Request signals from the 3 MBUS users and provides MBUS Grant signals to the appropriate user.The priority is a rotating type that ensures that each user has equal access to the bus based upon the previous user.

Author: LT Bryan Fetter, USN
Advisor: Dr. Russ Duren
Co-advisor: Dr. Hersch Loomis
Location: Naval Postgraduate School

Created: 25 October 2002
Modified: 7 November 2002
Simulated:
Target: XCV1000E FG1156
Software: Foundation 4.2i
Notes:


```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
--use IEEE.std_logic_unsigned.all;
--use IEEE.std_logic_arith.all;


package Grant is

component Grant_Logic
   port (
      M_Desire_Ext: in UNSIGNED (1 downto 0);
```

```
        M_Desire_Proc: in STD_LOGIC;
        M_Grant_Ext: out UNSIGNED (1 downto 0);
            M_Grant_Proc: out STD_LOGIC;
        Clk: in STD_LOGIC;
        Rst: in STD_LOGIC
    );
end component;

end package Grant;

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
--use IEEE.std_logic_unsigned.all;
--use IEEE.std_logic_arith.all;

entity Grant_Logic is
    port (
        M_Desire_Ext: in UNSIGNED (1 downto 0);
        M_Desire_Proc: in STD_LOGIC;
        M_Grant_Ext: out UNSIGNED (1 downto 0);
            M_Grant_Proc: out STD_LOGIC;
        Clk: in STD_LOGIC;
        Rst: in STD_LOGIC
    );
end Grant_Logic;

architecture Grant_Logic_arch of Grant_Logic is

type FSM_type is (Idle,Grant);
signal Curr_State, Next_State : FSM_Type;
signal User : UNSIGNED (1 downto 0);
signal Pri_0,Pri_1,Pri_2 : UNSIGNED (1 downto 0);

signal M_Desire_Int : UNSIGNED (2 downto 0);
signal M_Grant_Int : UNSIGNED (2 downto 0);


begin

M_Desire_Int(1) <= M_Desire_Ext(1);
M_Desire_Int(0) <= M_Desire_Ext(0);
M_Desire_Int(2) <= M_Desire_Proc;

M_Grant_Ext(1) <= M_Grant_Int(1);
M_Grant_Ext(0) <= M_Grant_Int(0);
```

```
M_Grant_Proc <= M_Grant_Int(2);


nxtStProc: process(Curr_State,Next_State, M_Desire_Int, User)

  begin

     case Curr_State is


       when Idle =>

         if M_Desire_Int /= "111" then
           Next_State <= Grant;
         else
           Next_State <= Idle;
         end if;

       when Grant =>
         if (M_Desire_Int(to_integer(User)) = '0') then
               Next_State <= Grant;
             else
               Next_State <= Idle;
             end if;

           when others =>
             null;

      end case;
    end process nxtStProc;

--Process to register current state

  curStProc: process (Clk, Rst)
  begin
    if (Rst = '0') then
       Curr_State <= Idle;
    elsif (Clk'event and Clk ='1') then
       Curr_State <= Next_State;
    end if;
  end process curStProc;

--Process to generate outputs

  outConProc: process(Curr_State,M_Desire_Int,Pri_0,Pri_1,Pri_2,User)
```

```
begin

  case Curr_State is

  when Idle =>
     M_Grant_Int <= "000";

     --to handle Reset
     if (Pri_0 = Pri_1) then
       if ((M_Desire_Int(0)) = '0' )then
         User <= "00";
       elsif ((M_Desire_Int(1)) = '0' )then
         User <= "01";
       elsif ((M_Desire_Int(2)) = '0' )then
         User <= "10";
       end if;
     elsif (M_Desire_Int(to_integer(Pri_0)) = '0' )then
       User <= Pri_0;
     elsif (M_Desire_Int(to_integer(Pri_1)) = '0' )then
       User <= Pri_1;
     elsif (M_Desire_Int(to_integer(Pri_2)) = '0' )then
       User <= Pri_2;
     end if;


   when Grant =>
     M_Grant_Int(to_integer(User)) <= '1';

     if User = "00" then
       Pri_0 <= "01";
       Pri_1 <= "10";
       Pri_2 <= "00";
     elsif User = "01" then
       Pri_0 <= "10";
       Pri_1 <= "00";
       Pri_2 <= "01";
     elsif User = "10" then
       Pri_0 <= "00";
       Pri_1 <= "01";
       Pri_2 <= "10";
     else
       Pri_0 <= "00";
       Pri_1 <= "01";
       Pri_2 <= "10";
     end if;
```

```
      when others =>
        null;

    end case;

  end process outConProc;

end Grant_Logic_arch;
```

==
XBUS Controller <xbus_controller.vhd>
==

Project: AYK-14 VHSIC Processor Module Hardware Emulator
Component: XBUS Controller
Description: State Machine that determines the user of the XBUS via use of the X_GRANT_LOGIC program and generates the control signals for XBUS operation depending upon type of operation and user. For I/O module (DSM) memory requests, generates the 23-bit address from Page Register set 0 and generates control signals for memory interface.

Author: LT Bryan Fetter, USN
Advisor: Dr. Russ Duren
Co-advisor: Dr. Hersch Loomis
Location: Naval Postgraduate School

Created: 25 October 2002
Modified: 21 November 2002
Simulated:
Target: XCV1000E FG1156
Software: Foundation 4.2i
Notes:

Disclaimer: NPS, makes no warranty for the use of this code or design. This code is provided "As Is". NPS, assumes no responsibility for any errors, which may appear in this code, nor does it make a commitment to update the information contained herein. NPS specifically disclaims any implied warranties of fitness for a particular purpose.


```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use WORK.X_GRANT.all;
use WORK.common.all;

package XBUS_CTRL is

component XBUS_Controller
   generic(
         FREQ: natural := 40_000-- operating frequency in KHz
   );
   port (
```

```
        Clk: in std_logic;
        Rst: in std_logic;
        -- Signals from Processor
        P_Command:      in unsigned(23 downto 0); --Command Word for X_BUS
        P_Data_In:      in unsigned(15 downto 0); --Data Word for X_BUS
        P_Data_Out:     out unsigned(15 downto 0);--Data read by XBUS
        --P_Page_0:                               --Page Register set 0
        P_Desire_L:     in std_logic;             --Desire Signal
        P_GRANT:        out STD_LOGIC;                  --Grant Signal

        -- Signals from Memory Arbitrator
        Mem_Addr:       out unsigned(22 downto 0);
        Mem_Data_WR:out unsigned(31 downto 0);
        Mem_Data_RD: in unsigned(31 downto 0);
        Mem_WR_Req: out std_logic;
        Mem_RD_Req:  out std_logic;
        Mem_Done:       in std_logic;
        --Test Port
        --Timer_Port:     out unsigned(1 downto 0);
        -- Signals on/off Adapter
        X_BUS:          inout unsigned(23 downto 0);
        X_GRANT_OUT:          out std_logic_vector(5 downto 0);
        X_DESIRE_IN_L:        in  std_logic_vector(5 downto 0);
        X_REQUEST_L:          inout std_logic;
        X_ACKNOWLEDGE_L:inout std_logic;
        X_RESUME_L: inout std_logic;
        IPC_MODE_L:  inout std_logic

    );
end component;

end XBUS_Ctrl;

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use WORK.X_GRANT.all;
use WORK.Common.all;


entity XBUS_Controller is
    generic(
                FREQ: natural := 40_000-- operating frequency in KHz
    );
    port (
        Clk: in std_logic;
```

```
    Rst: in std_logic;
    -- Signals from Processor
    P_Command:      in unsigned(23 downto 0); --Command Word for X_BUS
    P_Data_In:      in unsigned(15 downto 0); --Data Word for X_BUS
    P_Data_Out:     out unsigned(15 downto 0);--Data read by XBUS
    --P_Page_0:                               --Page Register set 0
    P_Desire_L:     in std_logic;             --Desire Signal
    P_GRANT:        out STD_LOGIC;                    --Grant Signal

    -- Signals from Memory Arbitrator
    Mem_Addr:       out unsigned(22 downto 0);
    Mem_Data_WR:out unsigned(31 downto 0);
    Mem_Data_RD: in unsigned(31 downto 0);
    Mem_WR_Req: out std_logic;
    Mem_RD_Req:  out std_logic;
    Mem_Done:       in std_logic;
    --Test Port
    --Timer_Port:     out unsigned(1 downto 0);
    -- Signals on/off Adapter
    X_BUS:          inout unsigned(23 downto 0);
    X_GRANT_OUT:            out std_logic_vector(5 downto 0);
    X_DESIRE_IN_L:          in  std_logic_vector(5 downto 0);
    X_REQUEST_L:            inout std_logic;
    X_ACKNOWLEDGE_L:inout std_logic;
    X_RESUME_L: inout std_logic;
    IPC_MODE_L:  inout std_logic
  );
end XBUS_Controller;

architecture XBUS_Controller_arch of XBUS_Controller is

--constants

constant DELAY_TWO_ZERO:       natural := 20;      -- 20 ns Delay interval
constant DELAY_FIVE_ZERO:      natural := 50;      -- 50 ns Delay interval
-- ACK Signal Max Delay (20ns)
constant TIMER_CYCLES_TWO_ZERO:   natural  :=  1  +  ((DELAY_TWO_ZERO  *
FREQ) / 1000000);
-- Delay (50 ns)
constant   TIMER_CYCLES_FIVE_ZERO:natural   :=   1   +   ((DELAY_FIVE_ZERO   *
FREQ) / 1000000);
constant MSTR_ADDR:        unsigned(3 downto 0) := "0000";
--Address of VPM on XBUS
--Constants for Clarity of Code
constant ACTIVE: std_logic := '1';
constant ACTIVE_L: std_logic := '0';            --For active low signal
```

```
constant INACTIVE: std_logic := '0';
constant INACTIVE_L: std_logic := '1';      --For active low signal

signal Timer, Timer_next: unsigned(log2(TIMER_CYCLES_FIVE_ZERO+1)-1 downto
0);
-- current Delay time
signal Time_Out:        std_logic;
signal User:            natural;

--All signals tied to input/output have same name with _int addended

signal  Clk_Int :       std_logic;
signal  Rst_Int :       std_logic;

signal  X_GRANT_OUT_Int:        std_logic_vector(6 downto 0);
signal  X_DESIRE_IN_L_Int:      std_logic_vector(6 downto 0);
signal  X_BUS_Int:          unsigned(23 downto 0);
signal  X_REQUEST_L_Int: std_logic;
signal  X_ACKNOWLEDGE_L_Int:std_logic;
signal  X_RESUME_L_Int:         std_logic;
signal  IPC_MODE_L_Int:         std_logic;


--Signal to drive INOUTS
signal Drive_X_BUS:         std_logic;
signal Drive_X_REQUEST:         std_logic;
signal Drive_X_ACKNOWLEDGE: std_logic;
signal Drive_X_RESUME:          std_logic;
signal Drive_IPC_MODE:          std_logic;


--Signals to Latch
signal P_Command_Int:           unsigned(23 downto 0);
--Command Word for X_BUS
signal P_Data_In_Int:       unsigned(15 downto 0); --Data Word for X_BUS
signal P_Data_Out_Int:          unsigned(15 downto 0); --Data Word for X_BUS
signal Mem_Data_WR_Int:         unsigned(31 downto 0);
signal Mem_Data_RD_Int:         unsigned(31 downto 0);
signal Mem_Done_Int:            std_logic;
signal Mem_Addr_Int:            unsigned(22 downto 0);

--Latch Driver Signals
signal P_Command_Latch:         std_logic; --Command Word for X_BUS
signal P_Data_In_Latch:         std_logic; --Data Word for X_BUS
signal P_Data_Out_Latch:    std_logic; --Data Word for X_BUS
signal Mem_Data_WR_Latch:       std_logic;
```

```
signal Mem_Data_RD_Latch:        std_logic;
signal Mem_Done_Latch:           std_logic;
signal Mem_Addr_Latch:           std_logic;



type FSM_type is (Idle,Proc_Bdcst,Req_Proc_Write,Ack_Proc_Write,Rsm_Proc_Write,
        Req_Proc_Read,Ack_Proc_Read,Read_Wait,Rsm_Proc_Read,
        DSM_Bdcst, Req_DSM_Write, Addr_ClkIn_DSM_WR, Ack_DSM_Write,
        Data_ClkIn_DSM_WR,Req_DSM_Read,Addr_ClkIn_DSM_RD,
        Data_ClkOut_DSM_RD, Ack_DSM_Read);
                --Proc_Bdcst Processor Broadcast Operation
                --Req_Proc_Write    Request Phase of Processor Write Operation
                --Ack_Proc_Write    Acknowledge Phase of Processor Write Operation
                --Write_Wait        Wait for resume signal to indicate memory written
                --Rsm_Proc_Write    Resume Phase of Processor Write Operation
                --Req_Proc_Read     Request Phase of Processor Read Operation
                --Ack_Proc_Read     Acknowledge Phase of Processor Read Operation
                --Rsm_Proc_Read     Resume Phase of Processor Read Operation
                --DSM_Bdcst         DSM Broadcast Operation
                --Req_DSM_Write     Request Phase of DSM Write Operation
                --Ack_DSM_Write     Acknowledge Phase of DSM Write Operation
                --Req_DSM_Read      Request Phase of DSM Read Operation
                --Ack_DSM_Read      Acknowledge Phase of DSM Read Operation

signal Curr_State, Next_State : FSM_Type;


begin

--Test Signal
--Test Port
--Timer_Port <= Timer;

--Connect all appropriate signals
Clk_Int <= Clk;
Rst_Int <= Rst;

X_DESIRE_IN_L_Int <= X_DESIRE_IN_L & P_Desire_L;
X_GRANT_OUT <= X_GRANT_OUT_Int(5 downto 0);
P_GRANT <= X_GRANT_OUT_Int(6);
Mem_Addr <= Mem_Addr_Int;
--X_RESUME_L_Int <= X_RESUME_L;
P_Data_Out <= P_Data_Out_Int;
Mem_Data_WR <= Mem_Data_WR_Int;
```

```
--Tristates for INOUTs

X_BUS <= X_BUS_Int when Drive_X_BUS = ACTIVE else (others =>'Z');
X_REQUEST_L <= X_REQUEST_L_Int when Drive_X_REQUEST = ACTIVE else
('Z');
X_ACKNOWLEDGE_L <= X_ACKNOWLEDGE_L_Int  when
Drive_X_ACKNOWLEDGE = ACTIVE else ('Z');
X_RESUME_L <= X_RESUME_L_Int when Drive_X_RESUME = ACTIVE else ('Z');
IPC_MODE_L <= IPC_MODE_L_Int         when  Drive_IPC_MODE  =  ACTIVE  else
('Z');

--Latch Signals
P_Command_Int <= P_Command when P_Command_Latch = ACTIVE else
P_Command_Int;
P_Data_In_Int <= P_Data_In when P_Data_In_Latch = ACTIVE else P_Data_In_Int;
P_Data_Out_Int <= X_BUS(15 downto 0) when P_Data_Out_Latch = ACTIVE else
P_Data_Out_Int;
Mem_Addr_Int <= X_BUS(22 downto 0) when Mem_Addr_Latch = ACTIVE else
Mem_Addr_Int;
Mem_Data_WR_Int <= ("0000000000000000" & X_BUS(15 downto 0))
        when Mem_Data_WR_Latch = ACTIVE else Mem_Data_WR_Int;
Mem_Data_RD_Int <= (Mem_Data_RD) when Mem_Data_RD_Latch = ACTIVE else
Mem_Data_RD_Int;
Mem_Done_Int <= Mem_Done when Mem_Done_Latch = ACTIVE else
Mem_Done_Int;


--Instantiate Grant Logic Module
 u0: X_GRANT_LOGIC port map (
                        X_Desire => X_DESIRE_IN_L_Int,
                                X_Grant => X_GRANT_OUT_Int,
                        X_Resume =>X_RESUME_L_Int,
                        Clk => Clk_Int,
                        Rst => Rst_Int
                            );


--Next State Conditioning Logic (Process 1)

nxtStProc:
process(Curr_State,Mem_Done,Timer,User,X_DESIRE_IN_L,X_RESUME_L,
         X_ACKNOWLEDGE_L,X_REQUEST_L,P_Command,
        X_BUS,X_GRANT_OUT_Int, Mem_Done_Int)

 begin
```

```
case Curr_State is

 when Idle =>
  if (X_GRANT_OUT_Int(6) = ACTIVE) then  --Processor Operations
    if (P_Command(19) = ACTIVE) then
      next_state <= Proc_Bdcst;
    elsif (P_Command(17) = ACTIVE) then
      next_state <= Req_Proc_Write;
    else
      next_state <= Req_Proc_Read;
    end if;
  elsif (X_GRANT_OUT_Int(5 downto 0) /= "000000") then  --DSM Operations
    if (X_REQUEST_L = ACTIVE_L) then
      if (X_BUS(19) = ACTIVE) then
        next_state <= DSM_Bdcst;
      elsif (X_BUS(19) = INACTIVE
            and X_BUS(23 downto 20) = MSTR_ADDR) then
        if (X_BUS(17) = INACTIVE) then
          next_state <= REQ_DSM_Read;
        elsif (X_BUS(17) = ACTIVE) then
          next_state <= REQ_DSM_Write;
        end if;
      end if;
    end if;
  else
    next_state <= Idle;
  end if;
  --Determine User
  if X_GRANT_OUT_Int(0) = ACTIVE then
    User <= 0;
  elsif X_GRANT_OUT_Int(1) = ACTIVE then
    User <= 1;
  elsif X_GRANT_OUT_Int(2) = ACTIVE then
    User <= 2;
  elsif X_GRANT_OUT_Int(3) = ACTIVE then
    User <= 3;
  elsif X_GRANT_OUT_Int(4) = ACTIVE then
    User <= 4;
  elsif X_GRANT_OUT_Int(5) = ACTIVE then
    User <= 5;
  else
    User <= 0;
  end if;
 --Broadcast Command by Processor
 when Proc_Bdcst =>
   if X_GRANT_OUT_Int(6) = INACTIVE then
```

```
    next_state <= Idle;
  else
    next_state <= Proc_Bdcst;
  end if;
--Processor Write Operations
when Req_Proc_Write =>
  if X_ACKNOWLEDGE_L = INACTIVE_L then
    next_state <= Ack_Proc_Write;
  else
    next_state <= Req_Proc_Write;
  end if;

when Ack_Proc_Write =>
  if X_RESUME_L = ACTIVE_L then
    next_state <= Rsm_Proc_Write;
  else
    next_state <= Ack_Proc_Write;
  end if;

when Rsm_Proc_Write =>
  if X_RESUME_L = INACTIVE_L then
    next_state <= Idle;
  else
    next_state <= Rsm_Proc_Write;
  end if;
   --Processor Read Operation

   when Req_Proc_Read =>
     if X_ACKNOWLEDGE_L = ACTIVE_L then
    next_state <= Ack_Proc_Read;
  else
    next_state <= Req_Proc_Read;
  end if;

   when Ack_Proc_Read =>
     if X_RESUME_L = ACTIVE_L then
    next_state <= Read_Wait;
  else
    next_state <= Ack_Proc_Read;
  end if;

when Read_Wait =>
  next_state <= Rsm_Proc_Read;

   when Rsm_Proc_Read=>
     if X_RESUME_L = INACTIVE_L then
```

```
  next_state <= Idle;
else
  next_state <= Rsm_Proc_Read;
end if;

 when DSM_Bdcst =>
   if (X_DESIRE_IN_L(User) = INACTIVE_L) then
  next_state <= Idle;
else
  next_state <= DSM_Bdcst;
end if;
 --DSM Write to Memory
 when Req_DSM_Write =>
   next_state <= Addr_ClkIn_DSM_WR;

 when Addr_ClkIn_DSM_WR =>
   if Timer = 0 then
     next_state <= Ack_DSM_Write;
   else
     next_state <= Addr_ClkIn_DSM_WR;
   end if;

 when Ack_DSM_Write =>
   next_state <= Data_ClkIn_DSM_WR;

 when Data_ClkIn_DSM_WR =>
   if (Mem_Done_Int = ACTIVE
      and X_DESIRE_IN_L(User) = INACTIVE_L) then
     next_state <= Idle;
   else
     next_state <= Data_ClkIn_DSM_WR;
   end if;
 --DSM Read from Memory
 when Req_DSM_Read =>
   next_state <= Addr_ClkIn_DSM_RD;

 when Addr_ClkIn_DSM_RD =>
   if Mem_Done = ACTIVE then
     next_state <= Data_ClkOut_DSM_RD;
   else
     next_state <= Addr_ClkIn_DSM_RD;
   end if;

 when Data_ClkOut_DSM_RD =>
   next_state <= Ack_DSM_Read;
```

```
          when Ack_DSM_Read =>
            if Timer = 0 then
              next_state <= Idle;
            else
              next_state <= Ack_DSM_Read;
            end if;

          when others =>
           null;

      end case;

--Timer Logic
     case Curr_State is

      when Idle =>
       null;

      when others =>

       if Timer /= TO_UNSIGNED(0,Timer'length) then
         Timer_next <= Timer - 1;
         Time_Out <= INACTIVE;
       else
         --Timer_next <= Timer;
         Time_Out <= ACTIVE;
       end if;
     end case;

   end process nxtStProc;

--Current State Vector Register (Process 2)

  curStProc: process (Clk_Int, Rst_Int)
  begin
   if (Rst_Int = '0') then
      Curr_State <= Idle;
      Timer <=  TO_UNSIGNED(0,Timer'length);
   elsif (Clk_Int'event and Clk_Int ='1') then
         Curr_State <= Next_State;
         Timer <= Timer_next;
   end if;
  end process curStProc;


--Output Conditioning Logic (Process 3)
```

```
outConProc: process(Curr_State,P_Command_Int,P_Data_In_Int,
                    Mem_Data_RD_Int)

begin

 --Default Signal to drive all Tristates High Z

     Drive_X_BUS <= INACTIVE;
     X_REQUEST_L_Int <= INACTIVE_L;
     Drive_X_REQUEST <= INACTIVE;
     X_ACKNOWLEDGE_L_Int <= INACTIVE_L;
     Drive_X_ACKNOWLEDGE <= INACTIVE;
     X_RESUME_L_Int <= INACTIVE_L;
     Drive_X_RESUME <= INACTIVE;
     IPC_MODE_L_Int <= INACTIVE_L;
     Drive_IPC_MODE <= INACTIVE;

 --Drive all outs inactive
   Mem_WR_Req <= INACTIVE;
   Mem_RD_Req <= INACTIVE;

 --Latch Drivers
     P_Command_Latch    <= INACTIVE; --Command Word for X_BUS
     P_Data_In_Latch    <= INACTIVE; --Data Word for X_BUS
     P_Data_Out_Latch <= INACTIVE;
     Mem_Data_WR_Latch<= INACTIVE;
     Mem_Data_RD_Latch<= INACTIVE;
     Mem_Done_Latch     <= INACTIVE;
     Mem_Addr_Latch     <= INACTIVE;

 case Curr_State is

  when Idle =>
   P_Command_Latch      <= ACTIVE;
   --This latches the signal when leaving Idle
   P_Data_In_Latch      <= ACTIVE;

  when Proc_Bdcst =>
   P_Command_Latch      <= ACTIVE;
   Drive_X_BUS <= ACTIVE;
   X_BUS_Int <= P_Command_Int;
   X_REQUEST_L_Int <= ACTIVE_L;
     Drive_X_REQUEST <= ACTIVE;

  when Req_Proc_Write =>
     Drive_X_BUS <= ACTIVE;
```

```
      X_BUS_Int <= P_Command_Int;
      X_REQUEST_L_Int <= ACTIVE_L;
         Drive_X_REQUEST <= ACTIVE;

    when Ack_Proc_Write =>
         Drive_X_BUS <= ACTIVE;
      X_BUS_Int(15 downto 0) <= P_Data_In_Int;
      Drive_X_REQUEST <= ACTIVE;

    when Rsm_Proc_Write =>
      Drive_X_REQUEST <= ACTIVE;

--Processor Read Operation

    when Req_Proc_Read =>
         Drive_X_BUS <= ACTIVE;
      X_BUS_Int <= P_Command_Int;
      X_REQUEST_L_Int <= ACTIVE_L;
         Drive_X_REQUEST <= ACTIVE;

    when Ack_Proc_Read =>
      Drive_X_REQUEST <= ACTIVE;

    when Read_Wait =>
         P_Data_Out_Latch <= ACTIVE;
         Drive_X_REQUEST <= ACTIVE;

    when Rsm_Proc_Read=>
         Drive_X_REQUEST <= ACTIVE;

--DSM Operations
    when DSM_Bdcst =>
         --No response Required

--DSM Write Operation
    when Req_DSM_Write =>
         Mem_Addr_Latch      <= ACTIVE;
      Drive_X_RESUME <= ACTIVE;
      Drive_X_ACKNOWLEDGE <= ACTIVE;

    when Addr_ClkIn_DSM_WR =>
         X_ACKNOWLEDGE_L_Int <= ACTIVE_L;
         Drive_X_ACKNOWLEDGE <= ACTIVE;
         Drive_X_RESUME <= ACTIVE;

    when Ack_DSM_Write =>
```

```
      Mem_Data_WR_Latch <= ACTIVE;
      Drive_X_RESUME <= ACTIVE;
      Drive_X_ACKNOWLEDGE <= ACTIVE;

    when Data_ClkIn_DSM_WR =>
      X_RESUME_L_Int <= ACTIVE_L;
         Drive_X_RESUME <= ACTIVE;
         Drive_X_ACKNOWLEDGE <= ACTIVE;
      Mem_WR_Req <= ACTIVE;

--DSM Read Operation
    when Req_DSM_Read =>
         Mem_Addr_Latch      <= ACTIVE;
         Drive_X_RESUME <= ACTIVE;
      Drive_X_ACKNOWLEDGE <= ACTIVE;

    when Addr_ClkIn_DSM_RD =>
         X_ACKNOWLEDGE_L_Int <= ACTIVE_L;
         Drive_X_ACKNOWLEDGE <= ACTIVE;
         Drive_X_RESUME <= ACTIVE;
         Mem_RD_Req <= ACTIVE;
         Mem_Data_RD_Latch <= ACTIVE;

    when Data_ClkOut_DSM_RD =>
         Drive_X_BUS <= ACTIVE;
      X_BUS_Int(15 downto 0) <= Mem_Data_RD_Int(15 downto 0);
         Drive_X_RESUME <= ACTIVE;
      Drive_X_ACKNOWLEDGE <= ACTIVE;

    when Ack_DSM_Read =>
      Drive_X_BUS <= ACTIVE;
      X_BUS_Int(15 downto 0) <= Mem_Data_RD_Int(15 downto 0);
      X_RESUME_L_Int <= ACTIVE_L;
         Drive_X_RESUME <= ACTIVE;
         X_ACKNOWLEDGE_L_Int <= INACTIVE_L;
      Drive_X_ACKNOWLEDGE <= ACTIVE;

    when others =>
     null;

   end case;

  end process outConProc;


end XBUS_Controller_arch;
```

==
Adapter Module <adapter_top.vhd>
==

Project:	AYK-14 VHSIC Processor Module Hardware Emulator
Component:	Adapter (Top level module)
Description:	Adapter module combines all of the components in the project, including the processor (data_path.vhd), and connects all appropriate signals. The ports correspond to the ports on the VPM and the SDRAM available on the AVNET board.
Author:	LT Bryan Fetter, USN
Advisor:	Dr. Russ Duren
Co-advisor:	Dr. Hersch Loomis
Location:	Naval Postgraduate School
Created:	25 October 2002
Modified:	1 December2002
Simulated:	
Target:	XCV1000E FG1156
Software:	Foundation 4.2i


```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use IEEE.std_logic_unsigned.all;
use WORK.common.all;
use WORK.Event_Bus.all;
use WORK.Add_Sel.all;
use WORK.Mem_Arb.all;
use WORK.Grant.all;
use WORK.oddParity.all;
use WORK.MBUS_CTRL.all;
use WORK.X_Grant.all;
use WORK.XBUS_CTRL.all;
use WORK.sdram.all;
```

```
entity Adapter_Top is
   generic(
                  SD_FREQ:     natural := 40_000;-- operating frequency in KHz
                  SD_DATA_WIDTH: natural := 16;-- host & SDRAM data width
                  SD_SADDR_WIDTH: natural := 12;-- SDRAM-side address width
                  SD_HADDR_WIDTH: natural := 23;
                  DATA_WIDTH_Arb: natural := 32;
                  ADDR_WIDTH_Arb: natural := 23;
                  XFREQ:         natural := 40_000
         );
   port (
         CLK:   in std_logic;
         RST:   in std_logic;
   --MBUS Signals
      M_BUS: inout unsigned(22 downto 0);
      --Handshaking Signals
      M_REQUEST_L: inout STD_LOGIC;
      M_ACKNOWLEDGE_L: inout STD_LOGIC;
      M_RESUME_L: inout STD_LOGIC;
      --Arbitration / Control Signals
      --M_DESIRE_OUT_L: out STD_LOGIC;
      M_DESIRE_IN_L: in unsigned(1 downto 0);
      M_GRANT_OUT: out unsigned(1 downto 0);
      --M_GRANT_IN: in STD_LOGIC;      --Used when VPM is slave
      M_BUSY_L: inout STD_LOGIC;
      S_BUSY_L: out STD_LOGIC;
      --MBus parity bits
      LSB_PARITY: inout STD_LOGIC;
      MSB_PARITY: inout STD_LOGIC;
      ADRS_PARITY: inout STD_LOGIC;
      CMD_PARITY: inout STD_LOGIC;
      --Control Bits
      MSB_WRITE_L: inout STD_LOGIC;
      LSB_WRITE_L: inout STD_LOGIC;
      BUS_ERROR_L: inout STD_LOGIC;
      THREE_TWO_DATA: inout STD_LOGIC;
      IPL_WRITE: inout STD_LOGIC;

     --XBUS Signals
      X_BUS: inout unsigned(23 downto 0);
      --Handshaking Signals
      X_REQUEST_L: inout STD_LOGIC;
      X_ACKNOWLEDGE_L: inout STD_LOGIC;
      X_RESUME_L: inout STD_LOGIC;
      --X_DESIRE_OUT_L: out STD_LOGIC;
      --Arbitration Signals
```

```
    X_GRANT_OUT: out std_logic_vector(5 downto 0);
    X_DESIRE_IN: in std_logic_vector(5 downto 0);
    --X_GRANT_IN: in STD_LOGIC;
    --O_X_GRANT_IN: in STD_LOGIC;
    --IPC Control
    IPC_MODE: inout STD_LOGIC;

   --Event System Signals
    E_BUS: in STD_LOGIC_VECTOR (7 downto 0);
    --Event Control Signals (EMON Bus)
    EMON: out STD_LOGIC_VECTOR (7 downto 0);

   --SDRAM Signals
      sclkfb:       in     std_logic;
      sclk:         out std_logic;
      sclk_tst:     out std_logic;
      cke:          out    std_logic;
      cs_n:         out    std_logic;
      ras_n:        out    std_logic;
      cas_n:        out    std_logic;
      we_n:         out    std_logic;
      ba:           out    unsigned(1 downto 0);
      sAddr:        out    unsigned(SD_SADDR_WIDTH-1 downto 0);
      sData:        inout unsigned(SD_DATA_WIDTH-1 downto 0);
      dqmh:         out    std_logic;
      dqml:         out    std_logic

  );
end Adapter_Top;

architecture Adapter_Top_arch of Adapter_Top is

signal  Clk_Int:        std_logic;
signal  Rst_Int:        std_logic;
--Signals for Event Controller
signal  E_VCTR_Int: std_logic_vector(8 downto 0);
signal  SR1_Bit_Int:    std_logic;
--Signals for Add_Select
signal  Add_In_Proc_Int:  unsigned (22 downto 0);
signal  Data_WR_Proc_Int: unsigned (31 downto 0);
signal  Data_RD_Proc_Int: unsigned (31 downto 0);
signal  RD_Req_in_Proc_Int: STD_LOGIC;
signal  WR_Req_in_Proc_Int: STD_LOGIC;
signal  Mem_req_Done_Proc_Int: std_logic;
    --MBUS Side
signal  Data_RD_MBUS_Int: unsigned (31 downto 0);
```

```
signal  Data_WR_MBUS_Int:  unsigned (31 downto 0);
signal  Add_out_MBUS_Int:  unsigned (22 downto 0);
signal  RD_Req_out_MBUS_Int: STD_LOGIC;
signal  WR_Req_out_MBUS_Int: STD_LOGIC;
signal  Proc_Desire_L_MBUS_Int: STD_LOGIC;
signal  Mem_req_Done_MBUS_Int:  STD_LOGIC;
        --OBM Side
signal  Add_In_OBM_Int: unsigned (22 downto 0);
signal  Data_RD_OBM_Int:  unsigned (31 downto 0);
signal  Data_WR_OBM_Int:  unsigned (31 downto 0);
signal  RD_Req_OBM_Int:  STD_LOGIC;
signal  WR_Req_OBM_Int:  STD_LOGIC;
signal  Mem_req_Done_OBM_Int: STD_LOGIC;
--Data Path
signal  IR_BUS_int_Int: std_logic_vector (31 downto 0);
signal  abs_addr_1_Int: std_logic_vector (22 downto 0);
signal  lcen_Int: std_logic;
signal  rcen_Int: std_logic;
signal  mem_READ_req_l_Int: std_logic;
signal  mem_WRITE_req_l_Int: std_logic;
--MBUS
signal  P_Grant_Out_Int: std_logic;
signal  M_Mem_Addr_Int:    unsigned(22 downto 0);
signal  M_Mem_Data_WR_Int: unsigned(31 downto 0);
signal  M_Mem_Data_RD_Int: unsigned(31 downto 0);
signal  M_Mem_WR_Req_Int: std_logic;
signal  M_Mem_RD_Req_Int: std_logic;
signal  M_Mem_Done_Int: std_logic;
--XBUS
signal  P_Command_Int:      unsigned(23 downto 0);
signal  P_Data_In_Int:      unsigned(15 downto 0);
signal  P_Data_Out_Int:     unsigned(15 downto 0);
signal  P_Desire_L_Int:     std_logic;
signal  P_GRANT_Int:        std_logic;
signal  X_Mem_Addr_Int:     unsigned(22 downto 0);
signal  X_Mem_Data_WR_Int: unsigned(31 downto 0);
signal  X_Mem_Data_RD_Int: unsigned(31 downto 0);
signal  X_Mem_WR_Req_Int: std_logic;
signal  X_Mem_RD_Req_Int: std_logic;
signal  X_Mem_Done_Int:     std_logic;
--SDRAM Ctrl
signal  SD_bufclk_Int: std_logic;
signal  SD_clk2x_Int:std_logic;
signal  SD_lock_Int: std_logic;
signal  SD_rd_Int: std_logic;
signal  SD_wr_Int: std_logic;
```

```
signal  SD_done_Int: std_logic;
signal  SD_hAddr_Int: unsigned(SD_HADDR_WIDTH-1 downto 0);
signal  SD_hDIn_Int: unsigned(SD_DATA_WIDTH-1 downto 0);
signal  SD_hDOut_Int: unsigned(SD_DATA_WIDTH-1 downto 0);
signal  SD_sdramCntl_state_Int: std_logic_vector(3 downto 0);




begin

--Clk_Int <= CLK;
Rst_Int <= RST;

        EBUS1: EVT_FSM port map(
                EBUS => E_BUS,
        CLK => Clk_Int,
        RST => Rst_Int,
        SR1_BIT => SR1_Bit_Int,     --Needs to be updated
        EMON => EMON,
        E_VCTR => E_VCTR_Int
        );

        ADD_SEL1: Add_Select port map(
                Add_In_Proc => Add_In_Proc_Int,
        Data_WR_Proc => Data_WR_Proc_Int,
              Data_RD_Proc => Data_RD_Proc_Int,
        RD_Req_in_Proc => RD_Req_in_Proc_Int,
              WR_Req_in_Proc => WR_Req_in_Proc_Int,
              Mem_req_Done_Proc => Mem_req_Done_Proc_Int,
    --MBUS Side
              Data_RD_MBUS => Data_RD_MBUS_Int,
              Data_WR_MBUS => Data_WR_MBUS_Int,
        Add_out_MBUS => Add_out_MBUS_Int,
              RD_Req_out_MBUS => RD_Req_out_MBUS_Int,
              WR_Req_out_MBUS => WR_Req_out_MBUS_Int,
              Proc_Desire_L_MBUS => Proc_Desire_L_MBUS_Int,
        Mem_req_Done_MBUS => Mem_req_Done_MBUS_Int,
    --OBM Side
        Add_In_OBM => Add_In_OBM_Int,
              Data_RD_OBM => Data_RD_OBM_Int,
        Data_WR_OBM => Data_WR_OBM_Int,
              RD_Req_OBM => RD_Req_OBM_Int,
              WR_Req_OBM => WR_Req_OBM_Int,
              Mem_req_Done_OBM => Mem_req_Done_OBM_Int
              );
```

```
Mem_Arb1: mem_arbitrator generic map(
        DATA_WIDTH => DATA_WIDTH_Arb,
        ADDR_WiDTH => ADDR_WIDTH_Arb)
        port map(
        Clk => Clk_Int,
        RST => Rst_Int,
        --Signals from SDRAM Controller
        Mem_Done => SD_done_Int,
        RD => SD_rd_Int,
        WR => SD_wr_Int,
        hAddr => SD_hAddr_Int,
        hData_In => SD_hDIn_Int,
        hData_Out  =>SD_hDOut_Int,
        --Signals from Processor
        P_Addr_In => Add_In_OBM_Int,
        P_Data_In => Data_RD_OBM_Int,
        P_Data_Out => Data_WR_OBM_Int,
        P_Mem_Done => Mem_req_Done_OBM_Int,
        P_RD => RD_Req_OBM_Int,
        P_WR => WR_Req_OBM_Int,
        --Signals from MBus
        M_Addr_In => M_Mem_Addr_Int,
        M_Data_In => M_Mem_Data_RD_Int,
        M_Data_Out => M_Mem_Data_WR_Int,
        M_Mem_Done => M_Mem_Done_Int,
        M_RD => M_Mem_RD_Req_Int,
        M_WR => M_Mem_WR_Req_Int,

        --Signals from XBus
        X_Addr_In => X_Mem_Addr_Int,
        X_Data_In => X_Mem_Data_WR_Int,
        X_Data_Out => X_Mem_Data_RD_Int,
        X_Mem_Done => X_Mem_Done_Int,
        X_RD => X_Mem_RD_Req_Int,
        X_WR => X_Mem_WR_Req_Int
        );

Processor:data_path port map(
        reset => Rst_Int,
        clock => Clk_Int,
        mem_req_DONE => Mem_req_Done_Proc_Int,
        mem_READ_req => RD_Req_in_Proc_Int,
        mem_WRITE_req => WR_Req_in_Proc_Int,
        IR_BUS => IR_BUS_Int,
        mem_BUS => Data_RD_Proc_Int,
```

```
            abs_addr => Add_In_Proc_Int,
            abs_addr_1 => abs_addr_1_Int,
            lcen => lcen_Int,
            rcen => lcen_Int,
            mem_READ_req_l => mem_READ_req_l_Int,
            mem_WRITE_req_l => mem_WRITE_req_l
            );

   MBUS: mbus_controller port map(
            Clk => Clk_Int,
   Rst => Rst_Int,
-- Signals from Processor
   P_Data_WR => Data_WR_MBUS_Int,
   P_Data_RD => Data_RD_MBUS_Int,
   P_Addr => Add_out_MBUS_Int,
   P_RD_Req => RD_Req_out_MBUS_Int,
   P_WR_Req => WR_Req_out_MBUS_Int,
   P_Desire_L => Proc_Desire_L_MBUS_Int,
   P_Mem_Done =>Mem_req_Done_MBUS_Int,
   P_Grant_Out => P_Grant_Out_Int,   --Grant signal to Processor

   -- Signals from Memory Arbitrator
   Mem_Addr => M_Mem_Addr_Int,
         Mem_Data_WR =>        M_Mem_Data_WR_Int,
   Mem_Data_RD =>     M_Mem_Data_RD_Int,
         Mem_WR_Req => M_Mem_WR_Req_Int,
         Mem_RD_Req => M_Mem_RD_Req_Int,
         Mem_Done => M_Mem_Done_Int,

         -- Signals on/off Adapter
         M_BUS => M_BUS,
         --M_GRANT_IN_L =>   ;          Used only when used as Slave
         M_DESIRE_IN_L => M_DESIRE_IN_L,
         M_GRANT_OUT =>     M_GRANT_OUT,
         --M_DESIRE_OUT_L ;--Used only when VPM used as Slave
         M_REQUEST_L =>      M_REQUEST_L,
         M_ACKNOWLEDGE_L => M_ACKNOWLEDGE_L,
         M_RESUME_L =>        M_RESUME_L,
         S_BUSY_L =>S_BUSY_L,
         M_BUSY_L =>M_BUSY_L,
         BUS_ERROR_L =>       BUS_ERROR_L,
            --Parity Bits
         LSB_PARITY => LSB_PARITY ,
         MSB_PARITY => MSB_PARITY ,
         ADRS_PARITY =>       ADRS_PARITY,
         CMD_PARITY =>        CMD_PARITY ,
```

```
        --Control Bits
      MSB_WRITE_L => MSB_WRITE_L ,
      LSB_WRITE_L =>       LSB_WRITE_L,
      THREE_TWO_DATA => THREE_TWO_DATA,
      IPL_WRITE => IPL_WRITE
      );

  XBUS: xbus_controller
        generic map(FREQ => XFREQ)
        port map (
  Clk => Clk_Int,
  Rst => Rst_Int,
        -- Signals from Processor
  P_Command => P_Command_Int,
  P_Data_In => P_Data_In_Int,
  P_Data_Out => P_Data_Out_Int,
  --P_Page_0:                              --Page Register set 0
  P_Desire_L => P_Desire_L_Int,
  P_GRANT => P_GRANT_Int,

  -- Signals from Memory Arbitrator
  Mem_Addr => X_Mem_Addr_Int,
      Mem_Data_WR => X_Mem_Data_WR_Int,
  Mem_Data_RD => X_Mem_Data_RD_Int,
      Mem_WR_Req => X_Mem_WR_Req_Int,
      Mem_RD_Req => X_Mem_RD_Req_Int,
      Mem_Done => X_Mem_Done_Int,

      -- Signals on/off Adapter
      X_BUS => X_BUS,
      X_GRANT_OUT => X_GRANT_OUT,
      X_DESIRE_IN_L => X_DESIRE_IN,
      X_REQUEST_L => X_REQUEST_L,
      X_ACKNOWLEDGE_L => X_ACKNOWLEDGE_L,
      X_RESUME_L => X_RESUME_L,
      IPC_MODE_L => IPC_MODE

);


  SDRAM: sdramCntl
        generic map(
        FREQ => SD_FREQ,
        HADDR_WIDTH => SD_HADDR_WIDTH,
        SADDR_WIDTH => SD_SADDR_WIDTH
        )
```

```
        port map (
        clkin => CLK,
        bufclk => SD_bufclk_Int,
        clk0 => Clk_Int,
        clk2x => SD_clk2x_Int,
        lock =>         SD_lock_Int,
        rst => Rst_Int,
        rd => SD_rd_Int,
        wr => SD_wr_Int,
        done => SD_done_Int,
        hAddr =>SD_hAddr_Int,
        hDIn => SD_hDIn_Int,
        hDout =>SD_hDOut_Int,
        sdramCntl_state => SD_sdramCntl_state_Int,
        -- SDRAM side
        sclkfb => sclkfb,
        sclk => sclk,
        sclk_tst => sclk_tst,
        cke => cke,
        cs_n => cs_n,
        ras_n => ras_n,
        cas_n => cas_n,
        we_n => we_n,
        ba => ba,
        sAddr => sAddr,
        sData => sData,
        dqmh => dqmh,
        dqml => dqml
    );


end Adapter_Top_arch;
```

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

1. Croskrey, M., *Design Recovery and Rapid Prototyping of a Legacy Processor,* Masters Thesis, Naval Postgraduate School, Monterey, CA, September 2002

2. "Aging Avionics in Military Aircraft," Committee on Aging Avionics in Military Aircraft, Air Force Science and Technology Board, Division on Engineering and Physical Sciences, National Research Council

3. Duren, Russ, "Options for Upgrading Legacy Avionics Systems," *Proceedings of the 21st Digital Avionics Systems Conference,* Irvine, CA, 27-31 October 2002

4. Datasegment.com. http://onlinedictionary.datasegment.com/word/Design%20recovery/ December 8, 1996.

5. Doom, Travis, *Formal Design Recovery for Obsolete Digital Systems,* Power Point Presentation, Write State University, Computer Science and Engineering

6. Chikofsky, E. and Cross II, J., Reverse Engineering and Design Recovery: A Taxonomy, January, 1990

7. Kidd, Christopher , Masters Thesis, Naval Postgraduate School, Monterey, CA, September 2002

8. Van den Bout, David, *The Practical XILINX Designers Lab Book*, Prentice_Hall, Inc., 1999

9. Rajan, Sundar, *Essential VHDL RTL Synthesis Done Right,*Sundar Rajan and Gennis Lafayette, 1999

10. Avnet Design Services, *Virtex-E Development Kit Users Manual*, Avnet Design Services, 2001

11. Micron, *256Mb: x4, x8, x16 SDRAM Industrial Temp,* Micron Technology Inc., 2002

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Chairman and Distinguished Professor Max F. Platzer, Code AA/PL
   Department of Aeronautics and Astronautics
   Naval Postgraduate School
   Monterey, California

4. Associate Professor Russell Duren, Code AA/DR
   Department of Aeronautics and Astronautics
   Naval Postgraduate School
   Monterey, California

5. Professor Herschel Loomis, Code EC/LM
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, California

6. Mr. Barry Douglas
   NAWC-WD
   F/A-18 Advanced Weapons Laboratory
   China Lake, California

7. Dr. Ken Trieu
   NAWC-WD
   F/A-18 Advanced Weapons Laboratory
   China Lake, California

8. Mr. Charles Bechtel
   NAWC-WD
   F/A-18 Advanced Weapons Laboratory
   China Lake, California

9. Mr. Rex Coombs
   PMA-209, Naval Air Systems Command
   NAS Patuxent River, Maryland

10. Commander Rich Brasel
U. S. Naval Test Pilot School
Naval Air Warfare Center Aircraft Division
NAS Patuxent River, Maryland